MASTER EN SCIENCES INFORMATIQUES

Etude de faisabilité de la génération automatique de commentaires à l'aide des techniques de Traduction Automatique Statistique

Jakubina, L

Award date:
2012

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix
Faculté d'Informatique
Année Académique 2011-2012

Étude de faisabilité de la génération automatique de commentaires à l'aide des techniques de Traduction Automatique Statistique

Laurent JAKUBINA

Mémoire présenté en vue de l'obtention du diplôme de Master en Sciences Informatiques

Résumé

Le développement logiciel existe depuis une cinquantaine d'années maintenant. En 50 ans, celui-ci a gagné en maturité au fur et à mesure que s'est constitué le patrimoine logiciel mondial. Durant cette évolution, le développement logiciel est passé par plusieurs phases, entre autres la crise du logiciel. Celle-ci est due à l'absence de méthodes de travail et de bonnes pratiques durant le processus de création de logiciels. C'est en réponse à cette crise qu'est né le génie logiciel. Le génie logiciel a appuyé l'importance de la documentation durant le développement pour notamment faciliter l'étape de la maintenance. Parmi cette documentation se trouve les commentaires écrits par les développeurs durant leurs activités d'implémentation. Cependant, malgré l'importance des commentaires, les développeurs se concentrent généralement sur l'efficacité de leur code source avant de le documenter. Afin de résoudre ce problème, certaines recherches visent à automatiser la génération de commentaires à partir du code source. Notre étude s'inscrit dans ce domaine. En effet, nous présentons une étude visant à évaluer la faisabilité de générer de façon automatique des commentaires à l'aide des techniques de traduction automatique statistique. Cette dernière est très en vogue aujourd'hui. La qualité des traductions générées par les traducteurs basés sur cette approche est impressionnante. L'exemple le plus connu est celui de Google, avec son outil de traduction en ligne Google Traduction. Cette approche consiste à générer des modèles probabilistes de traduction en se basant sur des corpus bilingues déjà existants. Purement statistique, l'approche offre l'avantage d'être indépendante vis-à-vis des langues du corpus, considérant les suites de phrases comme des séquences de caractères. Cet avantage nous permet d'entrevoir la possibilité de tester cette approche en créant un corpus composé de code source et de commentaires, puis d'y appliquer les outils de création de traducteurs automatiques. Finalement, exécuter une traduction à l'aide d'un traducteur généré en se basant sur un corpus code-commentaire reviendrait à générer des commentaires automatiquement en se basant sur le code source. C'est cette expérience que nous présentons dans ce mémoire.

Mots Clés : génie logiciel, traitement automatique du langage, ingénierie linguistique, développement logiciel, crise du logiciel, documentation, commentaire, génération automatique, maintenance, traduction automatique, traduction statistique, corpus bilingue aligné, bitexte, Moses.

Abstract

Software development has been around for about 50 now. During these fifty years, it has matured with the world-wide advancement of the software heritage. During this evolution, software development has been through many phases, including the software crisis. This crisis has been caused by the abscence of good working methods and coding habits during the software creation process. Software engineering has been born in response to this crisis. Software engineering has emphasised the importance of documentation during software development especially to ease maintenance. Among this documentation are the comments written by the developpers during the software implementation. However, despite the importance of comments, the developpers usualy focus on the efficiency of their code before documenting it. In order to solve this problem, some researches aim to automatise comment generation directly from the source code. Our study relates to this field. Indeed, we present a study which aims to evaluate the feasibility of automatically generating comments by using automatic statistical translation techniques. The latter is widely used today. The quality of translations generated by translators based on this approach is impressive. The most well-known example is Google's online translation tool : Google Translate. This approach consists in generating probabilistic translation models based on already existing bilingual corpora. Purely statistic, the approach offers the advantage of being independent towards the languages of the corpus considering sentences as being character sequences. This advantage lets us envision the possibility of testing this approach by creating a corpus composed of source code and comments, and then using automatic translator creation tools on it. In the end, executing a translation by using a translator generated from a code-commentary corpus amounts to automatically generating comments from the source code. This is the experiment we present in this memoir.

Key Words : software engineering, natural language processing, language engineering, software development, software crisis, documentation, comments, automatic generation, maintenance, automatic translation, statistical translation, aligned bilingual corpus, bitext, Moses.

Je tiens à remercier en premier lieu mon promoteur de mémoire, Naji Habra, pour son support, ses relectures et ses précieux conseils.

Je tiens à remercier également mes maîtres de stage, Sahraoui Houari et Langlais Philippe, pour leur accueil, leurs aides et leurs disponibilités durant mon stage à l'Université de Montréal.

Je remercie aussi énormément ma compagne, Chantal Hélène, et mon frère, Florian, pour leurs relectures minutieuses et leur soutien moral.

Merci finalement à mes parents pour leur soutien moral durant la rédaction de ce document.

À toutes ces personnes sans qui cette magnifique expérience à Montréal n'aurait jamais pu avoir lieu...

À tous, merci !

Table des matières

Table des figures

Chapitre 1

Introduction

1.1 Contexte

Situer les débuts du développement logiciel dans les années 50 semble un bon choix. C'est en effet durant cette période que naissent les trois premiers langages de programmation haut niveau que sont Fortran, Lisp et Cobol [Mati, 2012]. Plus proche du langage naturel et donc plus facilement accessible, s'en suit une certaine démocratisation de la création de logiciels. Le développement logiciel est lancé. Rapidement, les entreprises et les administrations réalisent les avantages de l'informatisation de leurs infrastructures, comme l'industrialisation en son temps. Traitements automatisés de l'information, gestions d'énormes quantités de données, les logiciels permettent à la fois de réduire considérablement les délais de traitement, mais aussi de réaliser des économies très significatives par rapport aux traitements manuels [Wiki, 2012f].

Petit à petit, c'est un véritable patrimoine logiciel qui se constitue. Patrimoine qui se complexifie avec le temps. Les nouvelles solutions logicielles cherchent à résoudre des problèmes de plus en plus complexes et il faut continuer à maintenir les solutions déjà existantes. Les conséquences ne se font pas attendre. Les produits réalisés ne se terminent pas dans les temps, coûtent plus cher que prévu, manque de fiabilité, etc. C'est la crise du logiciel [Fournier, 2012a, Fournier, 2012b, Hugues, 2002].

Cité pour la première fois en 1969, le génie logiciel vise à résoudre les problèmes de la crise du logiciel, et ceci en apportant des méthodes de travail et des bonnes pratiques afin de maîtriser le processus du développement logiciel [Fournier, 2012b, Hugues, 2002]. Parmi ces bonnes pratiques, le génie logiciel appuie l'importance de documenter le processus de développement du logiciel, et ce, au niveau même du code source, grâce à ce qu'on appelle communément "les commentaires" [Wiki, 2012b].

Malgré sa justification, l'étape de documentation est encore bien trop souvent bâclée, délaissée ou oubliée [Sridhara *et al.*, 2010]. La raison est simple. Du point de vue du développeur, mettre de côté l'implémentation pour en écrire la documentation semble être une perte de temps.

De plus, ils oublient l'étape de maintenance, étape complexe qui demandera plus tard à des développeurs n'ayant pas participé au développement initial du projet, de mettre à jour ce dernier.

1.2 Problématique

De ce constat sont nées les premières solutions de génération automatique de documentation [Wiki, 2012e]. Cependant, elles sont loins d'être parfaites. Surtout du point de vue du développeur, car ces dernières l'obligent toujours à devoir commenter son code source. La documentation étant constituée par extraction des commentaires et du code a proprement parlé. En conséquence, ces approches n'apportent pas non plus de solution aux éventuels plus vieux projets dont le code source n'avait pas été documenté. Pourtant, cela pourrait être une bonne chose en vue de l'étape de maintenance [Sridhara *et al.*, 2010].

Toutes ces raisons font que des recherches sont actuellement en cours afin de réaliser des outils qui seraient capables de documenter le code source d'un projet [Sridhara *et al.*, 2010, Rastkar *et al.*, 2011, Haiduc *et al.*, 2010], seulement en se basant sur ledit code source. C'est dans ce domaine de recherche que s'inscrit notre contribution.

1.3 Contributions

Ce mémoire présente l'expérience réalisée dans le but d'étudier la faisabilité de générer de façon automatique des commentaires de code source à l'aide des techniques de traduction automatique statistique.

La traduction automatique statistique est très en vogue aujourd'hui. La qualité des traductions générées par les traducteurs basés sur ces techniques sont impressionnantes [Goudet, 2012, Kouassi, 2009]. L'exemple le plus connu est le traducteur de Google, Google Traduction [1]. D'ailleurs, Google fourni une page web qui décrit son fonctionnement : `http://translate.google.fr/about/intl/fr_ALL`.

Ces techniques visent à effectuer des calculs statistiques, probabilistes et stochastiques sur des corpus bilingues, c'est-à-dire des ensembles de textes écrits dans une langue A, mais déjà aussi écrits/traduits dans une langue B. Indépendamment de la langue et de sa structure, l'approche vise à calculer les probabilités qu'un mot ou qu'une suite de mots du premier texte, sur base du nombre de répétitions et de leur position dans la phrase, soit le ou les mot(s) traduit(s) dans le deuxième texte. C'est ce que l'on appelle l'entraînement. L'obtention de probabilités permettant d'avoir des résultats impressionnants nécessite des volumes importants de textes traduits. En effet, plus il y a d'informations, meilleur est l'entraînement, meilleurs

1. `www.http://translate.google.com`

sont les probabilités. De ce fait, les corpus doivent se voir composer d'un minimum de quelques millions de mots.

L'aspect d'indépendance vis-à-vis de la structure de la langue fourni par l'approche permet d'imaginer la possibilité que la langue source soit le code source et la langue cible, les commentaires. Et finalement, que la traduction de la langue source vers la langue cible corresponde à une génération de commentaires sur base du code source.

L'implémentation d'outils réalisant ces techniques étant très complexe, notre contribution va proposer la composition d'un corpus bilingue code-commentaires sur lequel nous allons pouvoir exécuter les outils déjà existants et évaluer la faisabilité de l'idée de recherche.

1.4 Limites

Notre étude a des limites qu'il convient d'expliciter afin que le lecteur sache ce qu'il peut trouver dans ce document ou non. Ainsi, il ne s'agit pas de lister les limites rencontrées lors de notre étude. En effet, celles-ci sont présentées dans la section 10.3. Mais plutôt ce que notre étude n'a pas l'intention de développer.

- Selon le domaine de la gestion de projets, une étude de faisabilité est une étude qui tend à prouver qu'un projet est techniquement faisable et économiquement rentable, et ce, sur différents volets : technique, commerciale, économique, juridique et organisationnel [Wiki, 2012d]. Malgré l'usage du terme "Etude de faisabilité", notre expérience ne vise qu'à évaluer si l'idée de recherche est techniquement faisable. De ce fait, les autres volets ne sont pas abordés dans ce mémoire.
- Notre approche ne vise pas à modifier l'implémentation d'un des générateurs de traducteurs open source afin de lui faire prendre en compte les particularités des langages de programmation.

1.5 Structure du document

Ce mémoire se divise principalement en deux parties : La première, appelée *État de l'art* et la deuxième, notre *Étude de faisabilité*.

L'état de l'art apporte toutes les pièces nécessaires à compréhension de notre étude de faisabilité. Et ce, sur plusieurs points de vue : de sa justification dans l'histoire du développement logiciel jusqu'aux développements des principes qui sous-tendent l'idée de recherche [2]. En effet, le chapitre 2 commence par retracer l'histoire du développement logiciel, de ses débuts

2. L'approche générative par la Traduction Automatique Statistique.

en passant par la “crise du logiciel” afin de justifier la nécessité d’avoir une approche organisée face au développement logiciel : le Génie Logiciel. Ce dernier appuyant l’importance de la documentation durant le développement logiciel et vu que cette dernière est au centre de notre problématique, le chapitre 3 détaille l’étape de documentation. Notre recherche vise à automatiser l’étape de documentation. Cependant, nous ne sommes pas les premiers à traiter ce problème, ainsi le chapitre 4 explique ce qui existe déjà dans ce domaine. Finalement, le chapitre 5 décrit le fonctionnement de l’approche statistique de la traduction automatique et présente l’outil que nous utilisons dans notre étude afin d’atteindre notre objectif.

La deuxième partie est consacrée à l’expérience réalisée dans le cadre de notre étude de faisabilité. Le chapitre 6 présente plus en détail le principe et l’objectif de l’idée de recherche. Il structure aussi la suite de la lecture en fonction des différentes étapes de l’expérience. Ainsi le chapitre 7 explique en détail l’étape de préparation du corpus bilingue. Le chapitre 8 donne les moyens au lecteur de réaliser ses propres entraînements statistiques de corpus. Le chapitre 9, quant à lui, réalise les tests de traduction et les évaluations qui s’imposent. Finalement, le chapitre 10 conclut l’expérience en explicitant les résultats obtenus, les limites rencontrées et les éventuelles pistes de solutions.

Première partie

État de l'art

Chapitre 2

Génie Logiciel

L'informatique est une science jeune... très jeune. Là où certaines industries ont mis plus d'un siècle à s'installer, l'informatique s'est organisée en une cinquantaine d'années seulement. Devant cette vitesse de déploiement, l'industrie du logiciel s'est rapidement vue confrontée à un problème de taille : l'absence de méthodes, de techniques et d'outils dépassant largement le cadre de la programmation [Hugues, 2002]. C'est en réponse à ce problème et aux conséquences qu'il engendre qu'est né le génie logiciel. Cité pour la première fois lors de la conférence de l'OTAN à Garmisch en 1969 [Hugues, 2002, Fournier, 2012b], le génie logiciel s'est progressivement répandu dans l'industrie et dans les parcours d'apprentissage, apportant méthodes de travail et bonnes pratiques pour les acteurs du développement logiciel. C'est ainsi qu'est née l'instruction importante, donnée à tout apprenti programmeur, de documenter son code source. Afin de mieux comprendre l'importance du génie logiciel et ses apports, nous commençons par revoir les débuts du développement logiciel. Ensuite, nous expliquons les problèmes que l'informatique a connus à ses débuts, soit "la crise du logiciel". Finalement, nous présentons l'approche qu'est le génie logiciel dans le processus de développement des logiciels, en introduisant l'importance de la documentation, sujet du prochain chapitre.

2.1 Début du développement logiciel

Situer les débuts du développement logiciel n'est pas chose aisée. Comme beaucoup de technologies, l'informatique ne s'est pas développée en un jour. Ainsi, nous pourrions retourner assez loin dans le passé et y trouver des racines de notre informatique moderne.

Par exemple, voici un petit fait d'histoire concernant les langages de programmation :

> Les origines de ces codes remontent à l'an 820. Le mathématicien Al Khawarizmi publie à Bagdad un traité intitulé "La science de l'élimination et de la réduction", qui importé en Europe lors des invasions arabes aura une grande influence sur le développement des mathématiques. Plus tard, en 1840, le processus logique

> d'exécution d'un programme est appelé "algorithme", en l'honneur du mathématicien. Quinze années plus tard, Boole démontre que tout processus logique peut être décomposé en une suite d'opérations logiques appliquées sur deux états. [Mati, 2012]

De cette façon, nous pourrions parler de l'évolution des mathématiques à travers les siècles, de l'importance de la logique, d'Alan Turing, de la naissance des premiers ordinateurs, des cartes perforées, etc. Cependant, ce mémoire ne vise pas à retracer l'histoire entière de l'informatique. De ce fait, nous signalons seulement aux lecteurs curieux d'en savoir plus que de nombreuses ressources sont disponibles à ce sujet, notamment sur le web,.

Situer les débuts du développement logiciel dans les années 50 semble un bon choix. C'est en effet à cette époque que naissent notamment les trois premiers langages de programmation de haut niveau que sont Fortran, Lisp et Cobol [Mati, 2012]. Plus proche du langage naturel de l'homme et donc plus facilement accessibles, s'en suit une certaine démocratisation de la création des logiciels. Le développement informatique est lancé. Par la suite, des entreprises et des administrations réalisent les avantages qu'elles pourraient tirer de l'informatique : grâce aux capacités de calcul, aux traitements automatisés de l'information, aux possibilités d'enregistrer, de traiter et de restituer une grande quantité de données, l'informatique permet non seulement de réduire considérablement les délais de traitement, mais aussi de réaliser des économies très significatives par rapport aux traitements manuels [Wiki, 2012f].

À partir de là, ce sont des milliers de programmes qui sont réalisés, répondant aux besoins de gestion des entreprises [Wiki, 2012f] :

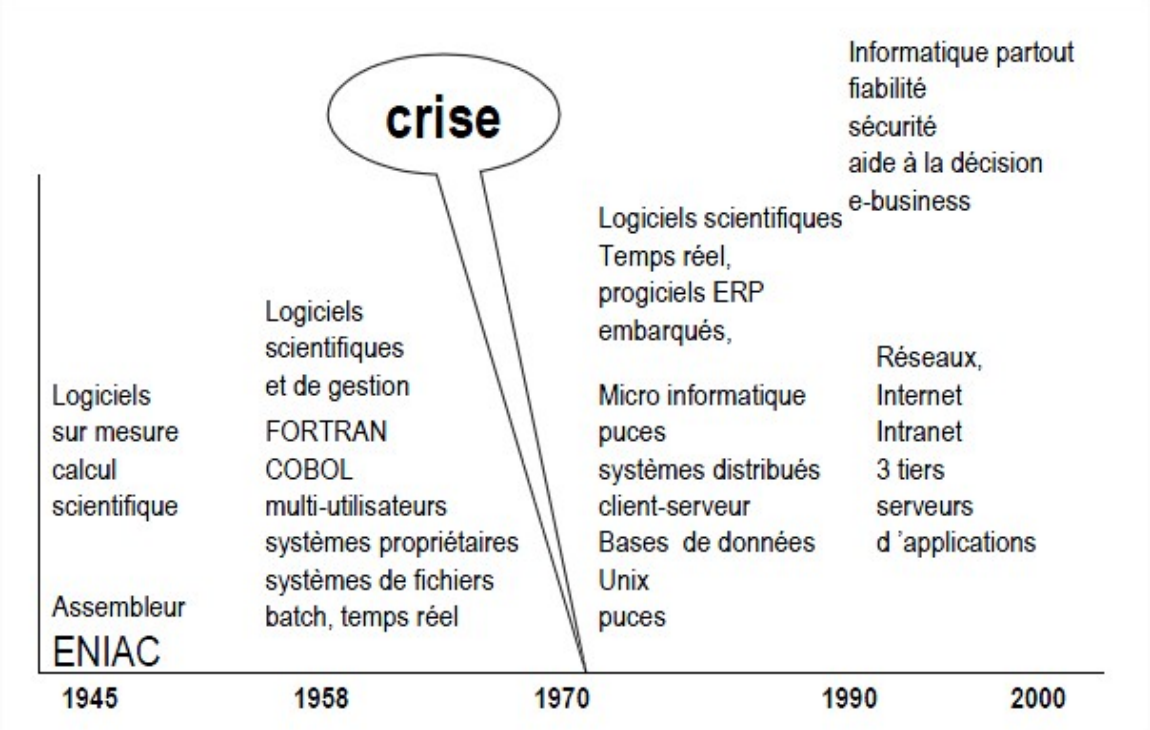
- La comptabilité : comptabilité générale, comptabilité analytique,...
- la facturation des clients et le règlement des fournisseurs,
- le suivi des comptes bancaires, la gestion de la trésorerie et la prévision financière,
- le paiement et le traitement social des salariés,
- la planification et le suivi de la production de l'entreprise,
- l'organisation et la mesure de l'efficacité commerciale,
- ...

Petit à petit, c'est un véritable patrimoine logiciel qui se constitue au sein des entreprises. Patrimoine qui va devenir de plus en plus complexe à gérer. Les lignes de code s'additionnent [1], l'entretien des programmes déjà existants et le développement de nouvelles solutions se complexifient. En conséquences, les produits réalisés ne sont pas terminés dans les temps, coûtent plus cher que prévu, ne sont pas fiables, peuvent être peu performants et coûtent cher à maintenir. Le réalisme de l'efficacité de l'informatisation frappe à la porte. C'est la crise du logiciel.

L'exemple le plus parlant, afin d'illustrer le manque total de prise en compte de la maintenance et des modifications futures des logiciels lors de cette génération, est sûrement l'existence du "fameux" bug de l'an 2000. Finalement pas aussi dévastateur qu'annoncé, le bug montre

1. Quelques millions de lignes de code pour certaines applications.

FIGURE 2.1 – 50 ans d'informatique



bien que les développeurs de l'époque n'avaient simplement pas prévu de voir leurs créations logicielles être encore utilisées en ce début de siècle [Wiki, 2012a].

2.2 Crise du logiciel

Nous sommes en 1969. Cela fait maintenant une quinzaine d'années que le développement logiciel dans le domaine industriel est lancé et le constat est sans appel [Fournier, 2012a] :

- Les logiciels réalisés ne correspondent pas souvent aux besoins des utilisateurs,
- les logiciels contiennent trop d'erreurs (qualité du logiciel insuffisante),
- les coûts du développement sont rarement prévisibles et sont généralement prohibitifs,
- la maintenance des logiciels est une tâche complexe et coûteuse,
- les délais de réalisation sont généralement dépassés,
- les logiciels sont rarement portables.

Les exemples de logiciels touchés par ces problèmes ne manquent pas [Fournier, 2012a, Habra, 2010] :

- Destruction par erreur de la fusée qui transportait la sonde Mariner 1 destinée à l'exploitation de Vénus (1962). À cause d'une erreur mineure dans une équation, l'ordinateur en a conclu un comportement bizarre de la fusée (ce qui n'était pas le cas) et en provoqua l'explosion [Fournier, 2012b].
- L'attaque simulée d'un missile américain alimente accidentellement un système de prévention (1979).

- Système de réservation SNCF Socrate, des retards catastrophiques (années 90).
- Panne du système téléphonique d'AT&T (1990).
- Crash d'Ariane V (1996) : "L'incident, dû à un bug dans les appareils informatiques utilisés par le pilote automatique, a provoqué la destruction de la fusée ainsi que la charge utile – 4 sondes de la mission Cluster – d'une valeur totale de 370 millions de dollars, ce qui en fait le bug informatique le plus coûteux de l'histoire." [Wiki, 2012h]
- (Événement belge) Erreur dans le comptage électronique après session de vote à Schaerbeek (2003).
- ...

et les études qui appuient ces problèmes non plus, comme le montre [Mireille,].

L'ensemble de ces problèmes sera dorénavant symptomatique de ce qui est appelé "La crise du logiciel". Ceux-ci sont à rapprocher d'un certain nombre de mythes, qui même aujourd'hui continuent de planer sur l'idée que se font les gens du développement logiciel [Fournier, 2012a] :

- Une idée grossière du logiciel à réaliser est suffisante pour commencer à écrire un programme (il est assez tôt de se préoccuper des détails plus tard). *Faux : une idée imprécise du logiciel à réaliser est la cause principale d'échecs.*
- Une fois que le programme est écrit et fonctionne, le travail est terminé. *Faux : la maintenance du logiciel représente un travail important : le coût de la maintenance représente d'ailleurs plus de 50% du coût total d'un logiciel.*
- Si les spécifications du logiciel à réaliser changent continuellement, cela ne pose pas de problèmes, puisque le logiciel est un produit souple. *Faux : des changements de spécifications peuvent se révéler coûteux et prolonger les délais.*
- Si la réalisation du logiciel prend du retard par rapport aux délais prévus, il suffit d'ajouter plus de programmeurs afin de finir dans les délais. *Faux : si l'on ajoute des gens à un projet, il faut compter une période de familiarisation. Le temps passé à communiquer à l'intérieur du groupe augmente également, lorsque la taille du groupe augmente. Cela réduit d'autant la productivité de chacun. L'introduction de nouvelles personnes dans un projet doit être étudiée et planifiée soigneusement.*

En d'autres termes, la raison de fond de la crise du logiciel réside donc plus dans le fait qu'il est beaucoup plus difficile de créer des logiciels que ce que le suggère notre intuition. En effet, les programmes, les données à traiter, les procédures,...étant essentiellement constitués de composants virtuels, on en sous-estime facilement la complexité [Strohmeier, 2012]. En conséquence, une série d'études se sont penchée sur la recherche de méthodes de travail adaptées à la complexité inhérente aux logiciels.

2.3 Nécessité d'une approche

Pour maîtriser la complexité des systèmes logiciels, il convient de procéder selon une démarche bien définie, de se baser sur des principes et méthodes, et d'utiliser des outils performants. En d'autres termes, de maîtriser le processus de développement des logiciels. C'est de cette volonté qu'est né le génie logiciel. Celui-ci cherche à établir et à utiliser des principes sains d'ingénierie dans le but de développer économiquement des logiciels qui sont fiables et qui fonctionnent efficacement sur des machines réelles [Strohmeier, 2012]. Ainsi, la recherche dans le domaine du génie logiciel est en perpétuelle évolution. Cependant, une série de normes de niveau international régisse aujourd'hui le domaine du génie logiciel.

Parmi celles-ci, nous retrouvons :
- Guide SWEBOK - Software Engineering Book of Knowledge
- ISO 12207 - Software life cycle
- IEEE Std 730 - IEEE Standard for Software Quality Assurance Plans
- IEEE Std 830 - IEEE Recommended Practice for Software Requirements Specifications
- IEEE Std 1002 - IEEE Standard Taxonomy for Software Engineering Standards
- IEEE Std 1063 - IEEE Standard for Software User Documentation
- ...

Ces normes décrivent le contenu du génie logiciel selon plusieurs points de vue. Ainsi, la norme 1002 présente le génie logiciel de deux façons différentes : d'après son contenu ainsi qu'en énumérant les activités typiques qui en font partie [Strohmeier, 2012]. La norme ISO 12207, quant à elle, présente le génie logiciel par l'étude de son cycle de vie. En effet, comme tout produit, un logiciel naît, vit et meurt. Durant toutes ces étapes, une série de méthodes et de bonnes pratiques accompagnent le processus logiciel comme le montre la figure 2.2.

Le développement du logiciel en tant que tel, étape principale du cycle de vie, est lui-même divisé en sous-étapes : l'analyse des besoins, la conception, l'implémentation, la validation et finalement, l'installation. De plus, l'ISO 12207 propose différentes façons d'agencer ces sous-étapes : les modèles de cycle de développements. Par exemple, le modèle en V (figure 2.3).

Les autres étapes, telles que la "Gestion de projet", l'"Évaluation de la qualité logicielle", la "Maintenance" et le "Développement de la documentation", généralement vues comme moins directement liées à la concrétisation du logiciel, n'en sont pas moins importantes. Au contraire même, l'expérience montre que la majeure partie du travail commence après la livraison du logiciel, soit lors de la maintenance. Rappelons que c'est aussi ce qu'a démontré la crise du logiciel.

Enfin, afin d'assurer un support correct et sur le long terme [2], n'est-il pas logique de mettre l'accent sur l'importance de la présence de la documentation, comme le suggère la norme ISO

2. Des dizaines d'années.

FIGURE 2.2 – Architecture du processus du logiciel

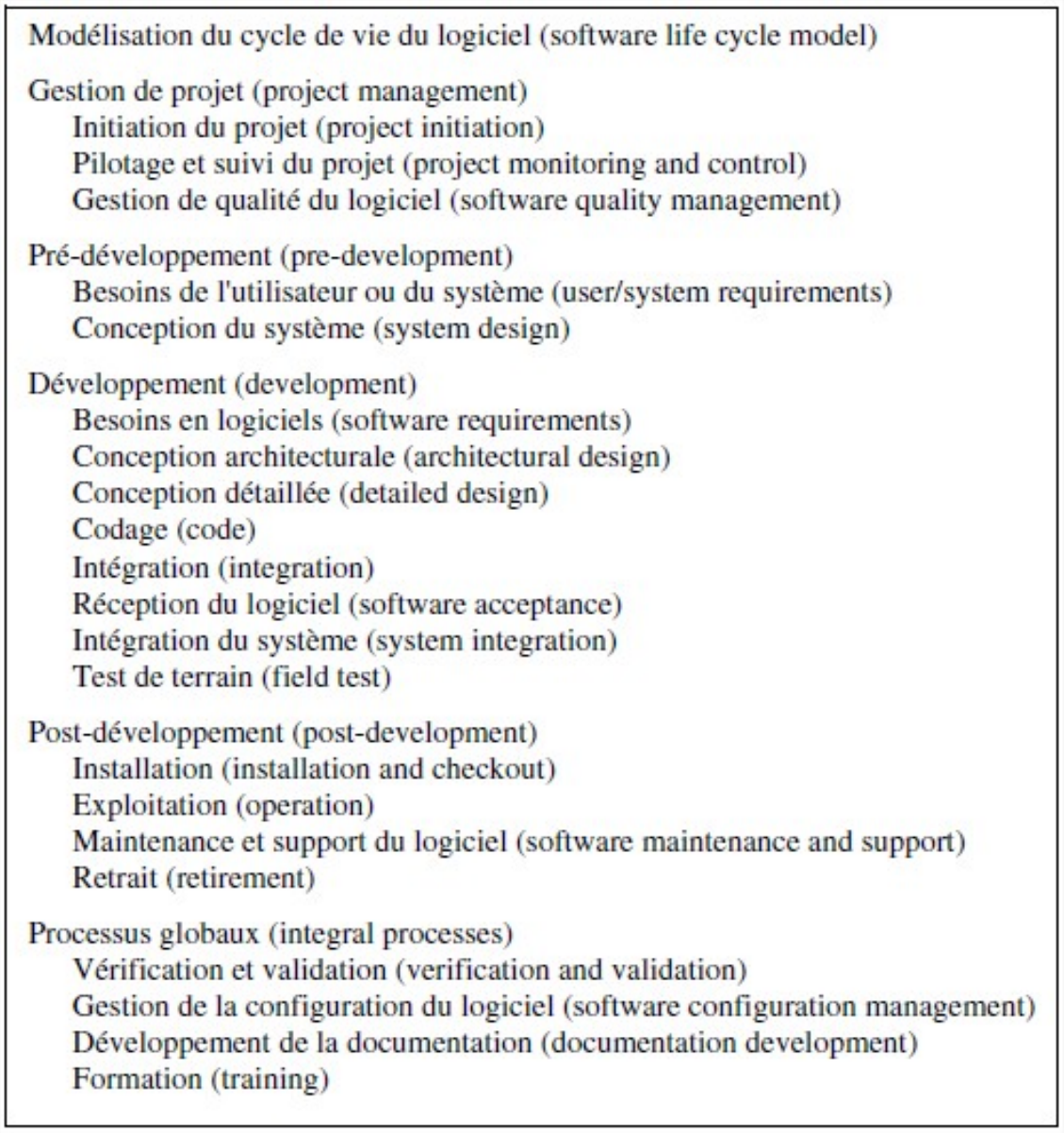
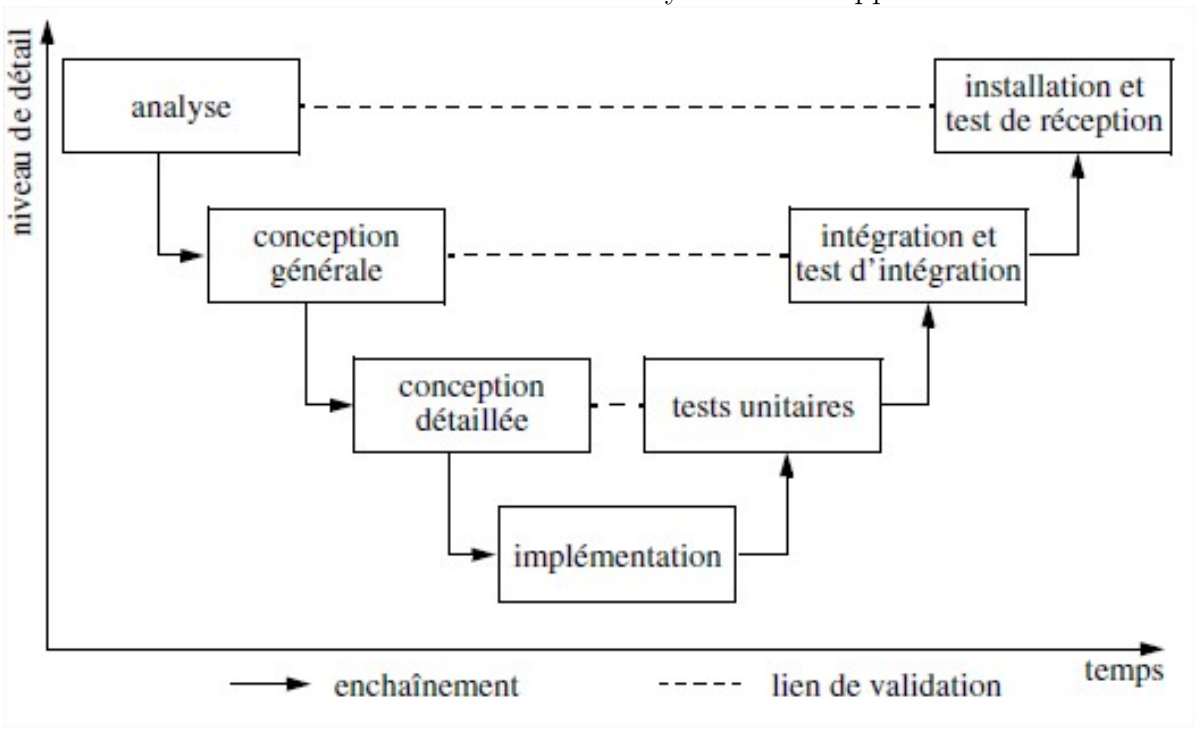


FIGURE 2.3 – Modèle en V du cycle de développement



12207, en plaçant celle-ci dans les processus globaux. Vu que celle-ci est au centre de notre recherche, la documentation est présentée en détail dans prochain chapitre.

Chapitre 3

Documentation

Le génie logiciel est apparu en réponse à la crise du logiciel, crise qui a mis en évidence les problèmes qu'engendrent le manque de maîtrise du processus de développement des logiciels. Parmi ceux-ci, la difficulté à maintenir les logiciels existants ainsi que la difficulté à gérer des projets de plus en plus complexes, sur lesquels peuvent travailler parfois plusieurs centaines de personnes.

Illustrons ces problèmes : n'importe quel développeur n'a pu échapper un jour à la surprise de ne plus comprendre un morceau de code...qu'il avait pourtant écrit lui-même quelque temps auparavant. En effet, malgré l'arrivée des langages de haut niveau durant les années 50, il n'en reste pas moins vrai que nous ne décrivons pas[1] *nos algorithmes en langage naturel et que de ce fait, la relecture du code source n'est pas toujours aussi simple que ce que l'on pourrait croire. Ce problème est d'autant plus vrai quand le code source a été écrit par une autre personne. Effectivement, malgré l'utilisation de plusieurs langages de programmation communs, la conception d'algorithmes, ou en d'autres termes, la résolution de problèmes, reste une activité intellectuelle dont les raisonnements vont être différents d'une personne à une autre. En conséquence, la maintenance de programmes se trouve être une des tâches les plus complexes du développement logiciel.*

Pour palier aux problèmes engendrés tels que la relecture complète du code source d'un projet, ou encore le temps passé à recomprendre un algorithme déjà implémenté[2]*, le génie logiciel a appuyé l'importance de l'étape de "Documentation", comme nous avons pu le voir dans le chapitre précédent, en la plaçant dans les processus globaux.*

Ce chapitre vise à présenter l'étape de documentation. Nous commençons par définir la notion de documentation ainsi que lister les différents types de documentations existantes durant le processus de développement d'un logiciel. En deuxième lieu, nous faisons un zoom sur un type de documentation en particulier, soit celle présente directement dans le code source, c'est-à-dire les commentaires. C'est en effet ceux-ci qui sont au centre de notre recherche. Finalement, nous

1. Pas encore ?
2. En vue d'une modification, d'une correction ou d'une optimisation par exemple.

terminons par un constat concernant l'étape de documentation, qui justifie en partie l'intérêt de la recherche effectuée dans le cadre de ce mémoire.

3.1 Définition et Types

> "La documentation, c'est la mémoire du projet. Elle permet de conserver la connaissance du projet, et elle permet de responsabiliser les acteurs. Elle permet aussi de juger de la conformité du produit réalisé avec les attentes." [piE, 2012]

La documentation logicielle est le texte écrit en langage naturel qui accompagne le logiciel informatique. Celle-ci a plusieurs objectifs. D'une part, elle explique comment le logiciel fonctionne et/ou comment doit-on l'employer. Deux niveaux d'explication sont disponibles : l'un est destiné aux utilisateurs du logiciel, tandis que l'autre est destiné aux programmeurs qui le conçoivent. Aussi, elle matérialise l'avancement des travaux, car chaque phase du développement est concrétisée par la production d'un ou plusieurs documents. Elle constitue aussi le support de communication entre les différents intervenants du projet [Wiki, 2012c, Strohmeier, 2012].

Il existe donc différents types de documentation, dépendant des acteurs ainsi que des étapes du processus du développement logiciel. Ainsi, la littérature distingue trois grands types de documentation [Wiki, 2012c, Strohmeier, 2012, F.Villeneuve, 2001, N.Rousse, 2004] :

- la documentation orientée "gestion de projets",
- la documentation dite technique,
- la documentation d'exploitation.

Chacune d'entre d'elles constituent d'autres sous-types de documentations, représentés notamment par des documents spécifiques, des plans, des manuels, etc.

Ainsi, sous le terme "gestion de projets" se retrouve toute la documentation concernant le planning du projet, la gestion du budget, le plan de développement du logiciel [3], un éventuel plan de qualité logicielle [4], l'analyse des besoins clients, etc. Concernant la documentation d'exploitation, ce sont les manuels d'utilisation, les manuels d'installation, des instructions éventuelles à propos du produit, etc. que nous retrouvons derrière cet intitulé. Finalement, la documentation dite 'technique" : cette dernière regroupe toute la documentation liée à l'implémentation du projet, c'est-à-dire la documentation du code source, des algorithmes, des interfaces de programmation, etc. En d'autres termes, la documentation du développeur. Cette documentation est habituellement incluse dans le code source lui-même de telle afin de la rendre facilement accessible pour quiconque serait amené à le traverser. En tant que développeurs, nous appelons cette activité "commenter son code source", ou encore "écrire ses commentaires".

Avec le cheminement effectué depuis le début du chapitre, le lien entre la maintenance et les commentaires est maintenant tracé. De cette façon, nous pouvons voir qu'il est important de

3. Cycle de vie.
4. Notons qu'un plan de qualité logicielle peut aussi exister pour la partie technique.

commenter le code source d'une application afin d'assurer sa maintenance. Cependant, malgré son importance, l'activité de commenter le code source est encore trop bien souvent bâclée et ignorée, comme nous l'expliquerons en fin de chapitre.

3.2 Les commentaires

Comme mentionnée dans la section précédente, une des premières formes de documentation logicielle est l'utilisation de commentaires. Placés au sein du code source et généralement repérables par un marqueur syntaxique particulier, ils représentent aussi le premier moyen de communication entre les développeurs. Du point de vue d'un compilateur ou d'un interpréteur, les commentaires sont des portions du code source à ignorer pour la machine. On peut donc y écrire ce que l'on veut. En pratique, un commentaire ressemble à une note écrite en langage naturel et accompagne un bout de code afin d'aider à la compréhension de ce dernier.

> "Ces petits morceaux de texte tentent désespérément d'éclaircir l'esprit du programmeur se replongeant dans un ancien code à faire évoluer." [Jibriss, 2012]

Plus précisément, commenter son code source a plusieurs utilités. Premièrement, s'aider soi-même. En effet, comme déjà mentionner durant l'introduction de ce chapitre, il est presque naturel pour n'importe quel développeur de ne plus comprendre une série d'instructions qu'il aurait lui-même codée quelque temps auparavant. En effet, rappelons qu'il ne suffit pas toujours de relire les lignes de code pour restituer la réflexion logique effectuée derrière celles-ci au moment de leur rédaction. De ce fait, écrire quelques lignes de commentaires pouvant résumer une explication ou donner une référence, permettra généralement d'éviter la perte de temps nécessaire à la re-compréhension de certaines parties du code source. Re-compréhension qui, rappelons le, est souvent à réaliser durant la maintenance.
Deuxièmement, commenter pour aider les autres. Il est devenu très rare aujourd'hui, au vu de la complexité des logiciels, d'être seul développeur d'une application. En d'autres termes, il y a d'énormes chances que les lignes de code qu'un développeur écrit doivent un jour être comprises par un autre dans une optique de maintenance ou d'optimisation. Ajoutant le fait que chaque développeur a son propre style et sa manière de coder, écrire quelques lignes de commentaires évitera à d'autres d'être complètement perdus à la lecture d'un chef d'œuvre d'implémentation.
Finalement, commenter en vue de former une documentation complète. En effet, certains outils, que nous verrons plus en détail dans un des chapitres suivants, ont la capacité de générer une documentation technique en dehors de l'implémentation. Utile à d'autres acteurs du processus de développement tels que les concepteurs, responsables de projets,...car elle leur évite de devoir plonger dans le code source, ces outils extraient des éléments de programmes importants du code source ainsi que leurs commentaires afin de constituer une documentation complète du logiciel.

Petite astuce qui découle directement de l'utilisation des commentaires. L'utilisation du marqueur syntaxique particulier des commentaires afin de cacher rapidement des portions de code au compilateur. Par exemple, pour tester des implémentations alternatives ou pour désactiver temporairement des fonctionnalités [Wiki, 2012b].

Avant de terminer cette section sur les commentaires, notons qu'il existe syntaxiquement parlant différents types de commentaires. Généralement caractérisés par un marqueur syntaxique particulier, certains diffèrent aussi en fonction du langage de programmation utilisé. En voici quelques exemples [Wiki, 2012b] :

- Les commentaires sur une seule ligne.
  - Ada, AppleScript, Lua, Lingo, Haskell : - - *commentaire*
  - (C99), C++, C#, D, Delphi (Pascal objet), Java, PHP, Scilab : // *commentaire*
  - Perl, Python, Ruby, C shell, Bourne shell, Tcl, et autres langages shell : # *commentaire*
- Les commentaires sur plusieurs lignes / dit en blocs.
  - C, C++, C#, CSS, Java, PHP et PL/I : /* *commentaire* */
  - Haskell : (* *commentaire* *)
  - SGML, HTML, XML : < !- - *commentaire* - ->

3.3 Constat

Nous avons retracé l'histoire du développement logiciel en mettant l'accent sur la crise que celui-ci a connue. De cette façon, le lecteur a pu saisir la justification de l'existence du génie logiciel et des avantages que celui-ci apporte : dans notre cas, la documentation. Arrivés là, nous pourrions croire que "tout va pour le mieux dans le meilleur des monde". Cependant, c'est loin d'être le cas. En effet, malgré les bonnes pratiques et méthodes de travail réfléchies et apportées par le génie logiciel, ce n'est pas pour autant que ces dernières sont appliquées unilatéralement par tous.

Premièrement, la programmation, science jeune et donc en manque de maturité, est encore fortement appliquée de façon "artisanale". Ceci est d'autant plus renforcé par le fait que, grâce aux ressources disponibles sur le web, "n'importe qui" aujourd'hui peut apprendre à programmer. Là n'est pas vraiment le problème, mais plutôt que la plupart de ces ressources ne font nullement référence aux bonnes pratiques du génie logiciel. En conséquences, pas mal de développeurs ont une approche relativement mal organisée face au développement logiciel. Et ce, indépendamment de leur talent en programmation.

Ensuite, même ceux qui suivent ou ont suivi un jour une formation en développement logiciel savent ce qu'ils ont pensé quand on leur a demandé les premières fois de documenter leur code : "Commenter son code, cela ne sert à rien et c'est une perte de temps !" ou encore "La documentation ? Je la ferai après...". En effet, malgré sa justification dans le processus de développement logiciel, l'activité de "documentation" semble en premier lieu être une perte

de temps. L'implémentation est mise de côté, donnant l'impression de ne pas faire avancer le projet. La formation et l'expérience viendront par la suite contrer cette impression.
Cependant, il n'en est pas moins vrai que, en cas de retard dans un projet, et ce, même dans le milieu industriel, il s'agit souvent des étapes les moins directement fonctionnelles, telles que l'écriture de la documentation, qui sont laissées tombées.

Le constat est ainsi sans appel. Malgré son importance, l'étape de documentation est encore bien trop souvent bâclée ou oubliée [Sridhara *et al.*, 2010]. Dans le but de pallier à ce problème, des recherches en génie logiciel sont effectuées afin d'aider l'ensemble des acteurs du développement logiciel dans la rédaction de la documentation. En effet, il existe aujourd'hui un certain nombre de techniques de génération automatique de documentation [Wiki, 2012e]. Vu que notre recherche s'axe sur ce principe, ces techniques sont présentées plus en profondeur dans le chapitre suivant.

Avant de passer au chapitre suivant, terminons par une petite note d'actualités. Comme nous venons de le voir par l'intermédiaire de ce constat, le génie logiciel a encore de la route à faire au sein du développement logiciel. Ainsi, n'oublions pas que celui-ci est né en réponse à la crise du logiciel. Alors qu'en est-il de cette dernière ? Pour un certain nombre d'auteurs, il n'est plus exact de parler de "crise du logiciel" mais bien de "maladie chronique du logiciel" [Strohmeier, 2012]. En effet, le développement logiciel est loin d'être maîtrisé à la perfection. . .et ce n'est pas l'article du Journal de Montréal du 30 juillet 2012 [5] qui va nous contredire.

5. Consultable en Annexes.

Chapitre 4

Génération Automatique de Documentation

Le chapitre précédent établit un constat : la documentation, malgré son importance dans le cycle de développement logiciel, reste encore bien souvent bâclée, délaissée ou oubliée. La maintenance s'en trouve complexifiée et nous retombons sur un des symptômes de la crise du logiciel. Afin d'éviter cette "catastrophe", il existe aujourd'hui un certain nombre d'approches de génération automatique de documentation, certaines relativement déjà bien installées dans les habitudes des développeurs, et d'autres actuellement à l'état de recherche. Vu que notre sujet de recherche s'inscrit dans la lignée de la génération automatique de documentation, ce chapitre présente ces différentes approches. Ainsi, nous commençons par réaliser une petite introduction au TAL, ou Traitement automatique du langage. Le TAL est un domaine qui relie à la fois l'informatique et la linguistique, autrement dit les langues naturelles. Étant donné que la génération automatique de documentation touche deux langues, notre langue naturelle et la "langue" de la programmation, c'est naturellement que nous verrons que les approches actuelles et en cours de recherches autour de la génération automatique de documentation découlent du TAL. Parmi celles-ci, la génération automatique par extraction, qui est actuellement utilisée par les générateurs de documentation actuels tels que JavaDoc ou Doxygen. Cette approche est présentée dans la deuxième partie. Finalement, la troisième partie laisse entrevoir les futures possibilités de génération automatique de documentation, via les recherches effectuées dans le domaine du TAL, comme c'est le cas pour la recherche effectuée dans le cadre de ce mémoire : la génération automatique de commentaires...par la traduction automatique statistique.

4.1 Traitement Automatique du Langage

Le Traitement Automatique du Langage, ou encore ingénierie linguistique est une discipline dont le champ se situe à la croisée des sciences du langage, de l'informatique et des sciences

cognitives [Deville, 2012]. Dans son acceptation la plus large, l'ingénierie linguistique concerne les applications informatiques capables de traiter le langage naturel, c'est-à-dire le langage parlé par les êtres humains par opposition aux langages artificiels, les langages de programmation. Le langage, parlé ou écrit, est le moyen de communication le plus naturel et le plus puissant dont l'homme dispose. Ainsi, depuis l'avènement des ordinateurs, le rêve existe de pouvoir un jour avoir des machines qui comprendraient le langage naturel, s'intégrant de cette façon à notre mode de fonctionnement avec la manière la plus ergonomique possible.

Pour cette raison, et d'autres aussi que nous ne détaillons pas [1], la recherche dans le traitement automatique de la langue a toujours suscité énormément d'intérêts tels qu'il en découle un certain nombre d'applications, certaines relativement bien connues et utilisées par l'utilisateur lambda d'un ordinateur [Wiki, 2012g] :

- la traduction automatique (c'est cette dernière qui a d'ailleurs lancé les recherches du TAL dans les années 50),
- la correction orthographique,
- la recherche d'information (et indirectement, la recherche sur le web),
- le résumé automatique de textes,
- la génération automatique de textes,
- la synthèse de la parole,
- la reconnaissance vocale,
- la reconnaissance de l'écriture manuscrite,
- ...

Chacune de ces applications ont un objectif différent. Cependant, les approches faites par chacunes d'entre elles autour de la langue s'entrecroisent énormément et s'aident mutuellement. Ainsi, l'approche "statistique" est impliquée aujourd'hui dans à peu près chaque application présentée ci-dessus, telle que certains parlent de "TAL Statistique" [Wiki, 2012g].

Nous ne développons pas toutes les approches qu'apporte la recherche dans le Traitement Automatique du Langage, cependant nous citons celles qui influencent ou pourront éventuellement influencer la génération automatique de documentation dans le cycle de développement du logiciel. Ainsi, les générateurs actuels génèrent une documentation par une approche d'extraction des éléments du code source. Vue plus en détail dans la prochaine section, cette approche par "extraction" est d'abord une des approches utilisées dans la génération automatique de textes et dans les résumés automatiques [PONTON, 1997, Danlos, , DELORT,]. Dans la continuité, des recherches d'une génération automatique de documentation d'un autre type, que nous verrons dans la troisième section, sont en cours en se basant notamment sur les théories réalisées dans le domaine de la génération automatique de résumés. D'ailleurs, notre approche s'inscrit aussi dans cette génération automatique d'un autre type en se basant sur la traduction automatique, application du TAL.

1. Le TAL est un domaine vaste tel que le résumer ferait l'objet de plusieurs mémoires.

En conclusion, nous voyons que logiquement, la génération automatique de documentation est fortement liée aux différentes applications du Traitement Automatique du Langage, laissant ainsi des portes ouvertes dans le domaine de la recherche afin d'aider les acteurs du développement logiciel lors de la rédaction de la documentation.

4.2 Approche par Extraction

L'approche par extraction est en fait l'approche utilisée par les générateurs de documentation actuels. En soi, ceux-ci ne "génèrent" par vraiment de la documentation ou, en d'autres termes, ils n'écrivent pas la documentation à partir du simple code source [2], comme aimerait l'espérer les développeurs. Cet espoir-là sera peut-être possible par les autres approches que nous présentons à la section 4.3.

Les générateurs de documentation actuels extraient certaines lignes du code source (exemple : les en-têtes de méthodes), les commentaires (marqués par un caractère syntaxique particulier) et parfois, certaines informations en provenance des fichiers binaires [Wiki, 2012e]. À partir de l'extraction, les outils réunissent ces informations et génèrent des manuels de référence au format HTML, PDF, etc. Avec ces derniers, nous avons donc la possibilité de consulter l'architecture d'un projet et les commentaires associés sans nécessairement aller consulter le code source.

Une série d'outils existent basés sur ce principe par extraction [Wiki, 2012e] :
- Javadoc (pour la langage Java)
- Doxygen (Multilangage mais principalement C)
- RDoc (pour le langage Ruby)
- ...

Les plus connus sont Javadoc et Doxygen. Afin de mieux voir en quoi consistent le principe et le résultat de la génération de documentation par cette approche, voyons un exemple à l'aide de l'outil pour le code Java : Javadoc.

4.2.1 Javadoc

Afin d'être affichés dans la documentation générée, les commentaires sont spécifiés par un style bien précis et utilisent les tags Javadoc. Premièrement, un bloc de commentaires est un bloc Javadoc quand il commence par le marqueur syntaxique suivant :"/**". Ensuite, les tags Javadocs sont des mots particuliers qui précèdent un commentaire qui doit être généré en retournant une information précise. Un tag Javadoc commence par un "@ [3]". En voici quelques exemples :

2. En d'autres termes, les commentaires doivent être écrits.
3. Arobase

FIGURE 4.1 – Code contenant des commentaires Javadoc

```
/**
 * Clean the comment by removing some special characters.
 * @param String comment
 * @return the comment without some special characters (/**, //, tab, @JavaDoc, etc.)
 * @author <a href="mailto:laurentjakubina@gmail.com">Laurent Jakubina</a>
 */
public static String cleanComment(String comment){
        comment = comment.replaceAll("/","")                        // Remove the char "/".
                    .replaceAll("\\*","")                           // Remove the char "*".
                        .replaceAll("\\n"," ")                      // Remove the new lines.

                          .replaceAll("@return", "RETURN ")

                            .replaceAll("(?m)@.*$","")              // Remove the Java-Doc lines.7
                                .replaceAll("(?m)JUnitDoclet.*$","")// Remove the JUnitDoclet comments
                                    .replaceAll("(?m)^\\s{1,}","")  // Remove the white space at the beginning of a line.
                                        .replaceAll("\\W"," ")
                                            .replaceAll("\\s{2,}"," ")
                                ;
        if (comment.matches("^\\W.*$") || comment.matches("^_.*$")){
            return "";
        } else
            return comment;
}

/**
 * Analyse the parents nodes of the comment (=noeudCourant) to know if the comment
 * describes a method, a class or some code.
 * @param Element noeudCourant
 * @return The type of the comment: CLASS, METHOD OR CODE as a string.
 * @throws JDOMException
 * @author <a href="mailto:laurentjakubina@gmail.com">Laurent Jakubina</a>
 */
public static String typeComment(Element noeudCourant)throws JDOMException{
```

– @author : Spécifie le nom du développeur qui a implémenté le code source.

– @return : Spécifie la valeur de retour d'une méthode/fonction.

– @version : Indique la version du code source.

– ...

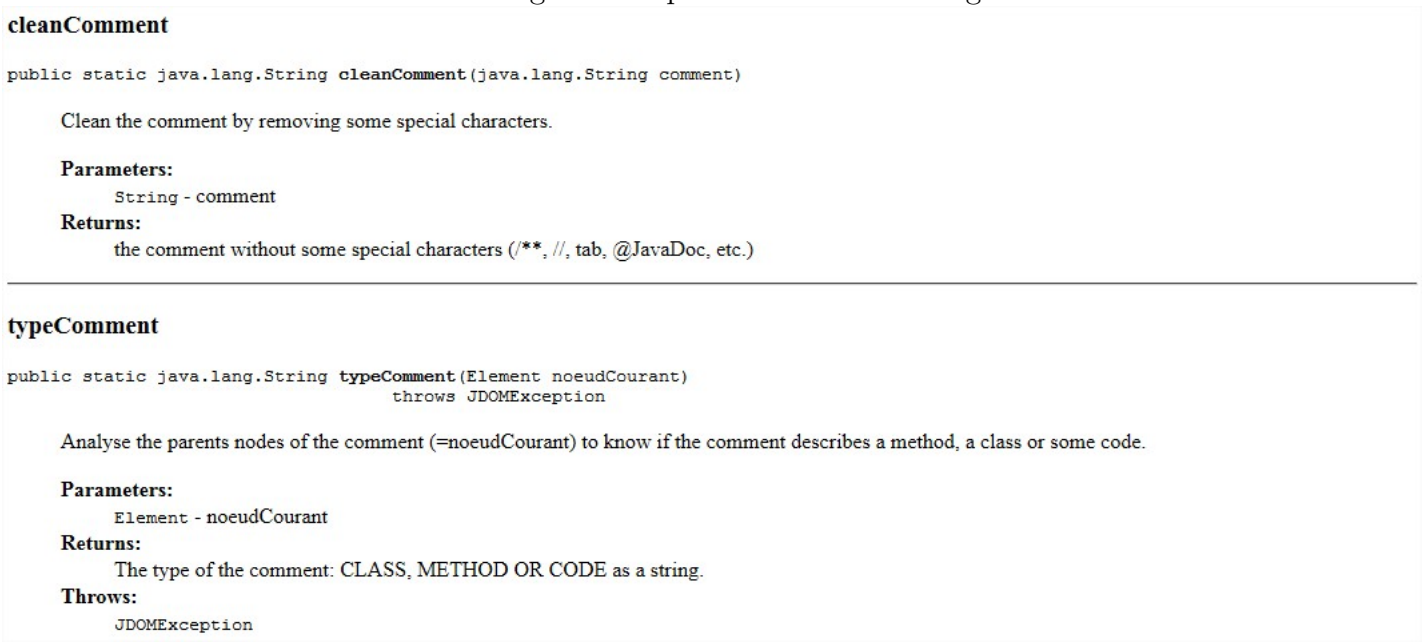
En utilisant ces mots-clés, le générateur est capable de préciser plus d'informations lors de la génération de la documentation. Précision : les commentaires Javadoc précèdent généralement les déclarations de classes, d'attributs et de méthodes. Un exemple de génération est présentée avec les figures 5.1 et 4.2.

4.3 Nouvelles Approches

L'approche par extraction et les outils actuels ont leurs avantages. Ainsi, ils permettent d'avoir une documentation complète du code source si évidemment, les commentaires sont écrits. Cette dernière est externalisée du code source en lui-même et peut donc être consultée indépendamment.

Le lecteur attentif aura remarqué la "problème" de cette approche. En effet, si le développeur ne réalise pas l'effort d'écrire les commentaires dans le code source, la documentation est en partie inexistante. Mais aussi, qu'en est-il des éventuels vieux projets ? N'y aurait-il pas un intérêt à documenter des éventuels vieux projets, qui sont en cours de maintenance par exemple,

FIGURE 4.2 – Javadoc générée à partir du code de la figure 5.1



cleanComment

```
public static java.lang.String cleanComment(java.lang.String comment)
```

Clean the comment by removing some special characters.

Parameters:
`String` - comment
Returns:
the comment without some special characters (/**, //, tab, @JavaDoc, etc.)

---

typeComment

```
public static java.lang.String typeComment(Element noeudCourant)
                                    throws JDOMException
```

Analyse the parents nodes of the comment (=noeudCourant) to know if the comment describes a method, a class or some code.

Parameters:
`Element` - noeudCourant
Returns:
The type of the comment: CLASS, METHOD OR CODE as a string.
Throws:
`JDOMException`

en évitant le travail de reconstitution de la documentation ? Pour parer à ces problèmes, des recherches autour d'une génération plus "vraie" sont en cours. Par génération plus "vraie", nous introduisons la possibilité de générer de la documentation à partir du code source, et du code source seulement. Ainsi, ces outils visent à générer de façon automatique des commentaires pour des méthodes, des fonctions, etc. Ce qui tend à se rapproche de notre recherche. Avant d'introduire le prochain chapitre en conclusion de celui-ci, voyons un peu de quoi retournent ces recherches en résumant très fortement trois articles scientifiques datant de 2010 et 2011.

Sridhara et al. [Sridhara *et al.*, 2010] décrit une nouvelle technique afin de générer automatiquement des commentaires résumés descriptifs de méthodes Java. Celle-ci prend en entrée le corps et la signature d'une méthode et en sort un résumé en langage naturel. Leur principe consiste à localiser dans le corps de la méthode les lignes de code les plus importantes, celles qui représentent les actions principales de la méthode, telles que des appels de méthodes ne retournant pas de valeurs (Void-Return S_Units), l'instruction consistant à sortir d'une méthode (Ending S_units), etc. Pour ces éléments, une phase appelée Text Generation détermine comment exprimer leur contenu sous forme de phrases en langages naturels. Finalement, ils combinent ces morceaux et les arrangent afin de produire un texte fluide (par exemple : éviter les éventuelles redondances). Par cette approche, basée uniquement sur le code source lui-même, ils peuvent générer des petits résumés de méthodes pour des projets parfois incomplets. Selon leurs résultats, les commentaires générés seraient précis, ne manqueraient pas les informations importantes et seraient raisonnablement concis.

Haiduc et al. [Haiduc *et al.*, 2010] vise à évaluer les techniques utilisées dans le domaine de résumés automatiques de textes afin de les adapter à la génération automatique de résumés de code source. Afin de pouvoir utiliser ces techniques, ils forment en premier lieu un corpus

de texte en langage naturel à partir du code source. Pour réaliser celui-ci, ils extraient les commentaires, les noms de méthodes Java qu'ils splittent en plusieurs mots et finalement, ils filtrent certains mots qui n'ont pas de significations particulières. C'est le cas des conjonctions de coordination, de certains verbes, pronoms, ...mais aussi des mots-clés du langage de programmation. Une fois le corpus créé, ils appliquent diverses techniques de résumés automatiques (Lead, VSM, LSI et baseline technique) sur ce dernier et analysent les résultats. En conclusion, ils déterminent que les meilleurs résultats sont obtenus en combinant certaines techniques de résumés automatiques et d'autres qui se basent sur la position des mots dans la phrase.

Rastkar et al. [Rastkar *et al.*, 2011] introduit une génération automatique de résumés en langage naturel pour les préoccupations transversales. Les préoccupations transversales sont des morceaux de code dispersés dans le code source, mais qui ont la particularité de toucher à un même aspect technique. Par exemple, le journalisation. Notons que la programmation orientée aspect vise à traiter ce problème. Afin d'aider le développeur à suivre les éléments de programme touché par ces préoccupations transversales, Rastkar et al. [Rastkar *et al.*, 2011] propose de produire automatiquement un résumé à partir d'une liste de méthodes qui sont liées par une préoccupation transversale. En premier lieu, ils extraient l'information structurelle et en langage naturel du code source. Deuxièmement, ils appliquent un ensemble d'heuristiques sur l'information extraite afin de trouver des patterns et des bouts de code touchés par une préoccupation transversale. Finalement, à partir des informations extraites et le contenu produit par l'heuristique, ils génèrent des phrases qui forment un résumé.

Parmi ces autres approches s'enligne aussi la recherche effectuée dans le cadre de ce mémoire : générer de façon automatique des commentaires, à l'aide de la traduction automatique statistique. Comme nous venons de le voir, la traduction automatique et l'approche statistique proviennent aussi du domaine du Traitement Automatique du Langage. Afin de bien comprendre l'approche réalisée dans le cadre de notre recherche, la traduction automatique et son aspect statistique sont présentés dans le prochain chapitre.

Chapitre 5

Traduction Automatique Statistique

Le chapitre précédent a présenté différentes façons de générer de la documentation logicielle, et ce, de façon automatique. Nous avons conclu en expliquant brièvement que l'approche explorée dans le cadre de ce mémoire était la génération de commentaires grâce aux techniques de la traduction automatique statistique. Mais qu'est-ce que la traduction automatique statistique ? C'est ce que ce chapitre tente d'expliquer.
Nous commençons par une petite préquelle, qui explore rapidement d'où vient, ce besoin de traduction pour l'homme. Ensuite, nous détaillons cette branche du Traitement Automatique du Langage qu'est la Traduction Automatique. En effet, après la définition et un peu d'histoire, nous expliquons les différentes approches qui composent la théorie de la Traduction Automatique. Parmi celles-ci, l'approche statistique qui est celle exploitée dans notre recherche. Cette approche est expliquée en détail dans la troisième partie de ce chapitre. Finalement, afin que le lecteur ait toutes les cartes en main pour comprendre la deuxième partie de ce mémoire, nous présentons brièvement les outils qui permettent d'appliquer l'approche statistique, en accordant une attention toute particulière à Moses, la boîte à outils que nous utilisons lors de notre étude de faisabilité.

5.1 Préquelle

Depuis les débuts de son existence, l'homme vit à travers l'acte de communication. Il lui est d'ailleurs presque indispensable de communiquer afin d'assurer sa survie. Son expansion à travers le monde l'a amené à s'éparpiller et en conséquence, les proto-langues ont peu à peu évolué en de très nombreuses langues différentes. C'est ainsi que naquit rapidement le besoin de traduction.

La traduction implique de transformer une séquence de mots appartenant à une langue source en une séquence de mots dans une langue cible, tout en conservant le sens original de cette séquence [Monty, 2010]. Cette activité est loin d'être simple et nécessite un certain temps

d'apprentissage. La possibilité d'automatiser ce processus peut donc être vue comme un énorme pas en avant pour l'humanité.

Les premiers efforts d'automatisation datent de 1933. Artsrouni et Trojanskij, respectivement de France et de Russie, développèrent indépendamment l'un de l'autre des machines qui servaient, entre autres, de dictionnaires mécaniques, mais aussi à la traduction [Monty, 2010].

Cependant, c'est après la Seconde Guerre mondiale que l'idée se fait jour d'une machine à traduire. Les premiers ordinateurs, comme l'Eniac, esquissent cette possibilité, mais aussi le sentiment qu'effacer la barrière des langues pourrait contribuer à la paix universelle [Goudet, 2012]. Un énorme engouement se lance alors derrière les possibilités de Traduction Automatique.

5.2 Traduction Automatique

5.2.1 Définition

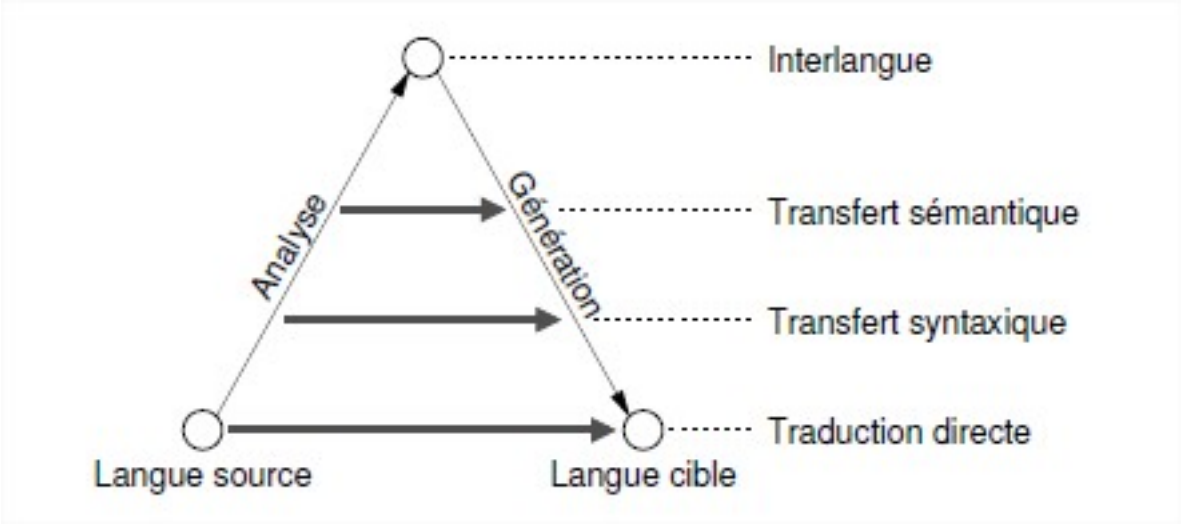
L'expression "Traduction Automatique", ou TA désigne l'activité de transcription de messages d'une langue naturelle (ex : le Français) vers une autre (ex : l'Anglais), et ce, de façon entièrement automatisée, c'est-à-dire sans intervention humaine. Notons que le terme "Traduction Automatique" réfère aussi à un sous-domaine de l'ingénierie linguistique, comme déjà listé dans la section 4.1. Cette branche du TAL travaille sur la théorie de la traduction automatique et à la mise en œuvre d'outils réalisant ces théories.

5.2.2 Histoire

L'histoire de la TA est longue et complexe. En effet, comme le TAL à la section 4.1, la traduction automatique constitue une des premières applications visant à être automatisées dès la naissance des ordinateurs, après la Deuxième Guerre mondiale. De ce fait, le lecteur est invité à lire d'autres ressources telles que [Hutchins, 1995] afin d'en savoir plus.

Cependant, nous pouvons diviser son histoire en deux parties, basée sur la dominance des différentes approches que la TA a connues : avant 1990 et après. Ces dernières seront vues par la suite.
Ainsi, de 1950 à 1990, l'approche qui a dominé la TA est la "Traduction Automatique à base de règles". Celle-ci repose sur l'utilisation de règles linguistiques et d'entrées de dictionnaires pour chaque paire de langues [Systran, 2012]. Sur les 40 ans, la TA à base de règles a grandi sur base de l'évolution des règles linguistiques, ce qui l'a conduite à être divisés en sous-approches que nous explicitons dans la section 5.2.3.
Le début des années 1990 a vu le développement d'autres types d'approches. Les ordinateurs se répandent et gagnent en puissance, ce qui permet l'émergence de stratégies qui se fondent sur

FIGURE 5.1 – Le triangle dit “de Vauquois”



de grandes quantités de données : la “Traduction Automatique à base de corpus”. Comme pour la TA à base de règles, on distingue encore deux sous-approches : la traduction automatique à base d’exemples et la traduction automatique par méthodes statistiques.

5.2.3 La TA à base de règles

Avant 1990, les recherches en Traduction Automatique tournaient autour de la conception de règles linguistiques sophistiquées et de dictionnaires gigantesques. En effet, il semble logique de traiter un processus “linguistique” à partir de composantes linguistiques tel que nous le faisons nous, c’est-à-dire à partir de vocabulaire (ex : un dictionnaire) et de règles (ex : règles de grammaires, de syntaxes, etc.)

C’est ainsi qu’en 1968, Vauquois propose une analyse de processus de traduction, encore pleinement pertinente et employée de nos jours. Sa représentation synthétique se fait sous la forme d’un triangle : Le triangle dit “De Vauquois” [Déchelotte, 2007, Haithem, 2010]. La traduction peut s’opérer à plusieurs niveaux : directe, par transfert ou grâce à une langue “pivot”, comme le montre la figure 5.1.

La traduction directe

La traduction directe se base sur l’équivalence des termes. Ainsi, à partir de la consultation d’un dictionnaire, les mots sont traduits de la langue source vers les mots de la langue cible, sans aucune analyse. C’était l’approche utilisée par les systèmes de traduction de premières générations des années 50. Utile, mais évidemment très limitée [Haithem, 2010].

En effet, pour traduire, le sens d’un texte original (source) doit être compris pour être restitué dans la langue cible. La traduction ne doit donc pas se limiter une simple substitution mot à mot. Le traducteur doit analyser et interpréter le texte et comprendre les relations entre

les mots qui peuvent influencer son sens. Ceci requiert une connaissance de la grammaire, de la syntaxe (structure de la phrase) et de la sémantique (sens des mots) à la fois dans la langue source et dans la langue cible [Systran, 2012].

La traduction par transfert

Comme le montre le triangle de Vauquois, l'approche par transfert vise à analyser le texte source du point de vue syntaxique et sémantique. De ces analyses découle une série de règles linguistiques, qui ont pour but d'être les représentations abstraites des connaissances grammaticales, syntaxiques et sémantiques. Le transfert entre la langue source et la langue cible peut être effectué de façon descendante ou ascendante opérant ainsi à deux niveaux différents de représentation entre les deux langues [Haithem, 2010].

La traduction par langue pivot

L'analyse totale du texte source peut aboutir à une représentation de son sens dans une "interlangue" artificielle [Déchelotte, 2007]. L'approche reposant sur une interlangue est séduisante, car elle remplace le problème de la traduction par deux problèmes monolingues, l'analyse et la génération. L'importance de cette approche vient du fait que les modules d'analyse et de génération sont réutilisables pour la création d'un système pour un nouveau couple de langues [Haithem, 2010].

5.2.4 La TA à base de corpus

La TA par méthodes statistiques

L'approche statistique de la traduction automatique étant celle utilisée dans notre expérience, celle-ci est expliquée en détail dans la section 5.3.

La TA à base d'exemples

La traduction automatique à base d'exemples repose sur l'utilisation d'un ensemble de textes préalablement traduits : un corpus parallèle, où chaque phrase dans la langue source est mise en parallèle avec sa traduction dans la langue cible.

Lorsqu'on lui présente une phrase à traduire, le système parcourt sa base d'exemples et produit trivialement une traduction si la phrase s'y trouve. Dans le cas général, la phrase n'apparaît pas dans la base et le système s'emploie alors à rassembler des exemples qui contiennent des fragments communs (des groupes de mots) avec la phrase à traduire. Pour chaque fragment d'exemple dans la langue source, il s'agit ensuite de retrouver sa traduction dans la langue

cible : c'est la phase d'alignement. Enfin, la phase de génération assemble les fragments dans la langue cible et produit la traduction [Déchelotte, 2007].

5.2.5 Comparaison

TA à base de règles	TA à base de corpus
Qualité contrôlable : Traductions prévisibles et homogènes	Traductions imprévisibles et hétérogènes
Bonne qualité des traductions généralistes	Qualité faible sur les domaines généralistes
Connaissance des règles grammaticales	Pas de connaissance de la grammaire
Performance et robustesse	Besoin en CPU et espace disque importants
Cohérence entre les versions	Pas de cohérence entre les versions
Manque de fluidité	Traductions fluides
Difficulté à gérer les exceptions	Reconnaît bien les exceptions
Coût de développement et de personnalisations élevés	Développement rapide et peu onéreux si des corpus de qualité sont disponibles

[Systran, 2012]

5.2.6 Conclusion

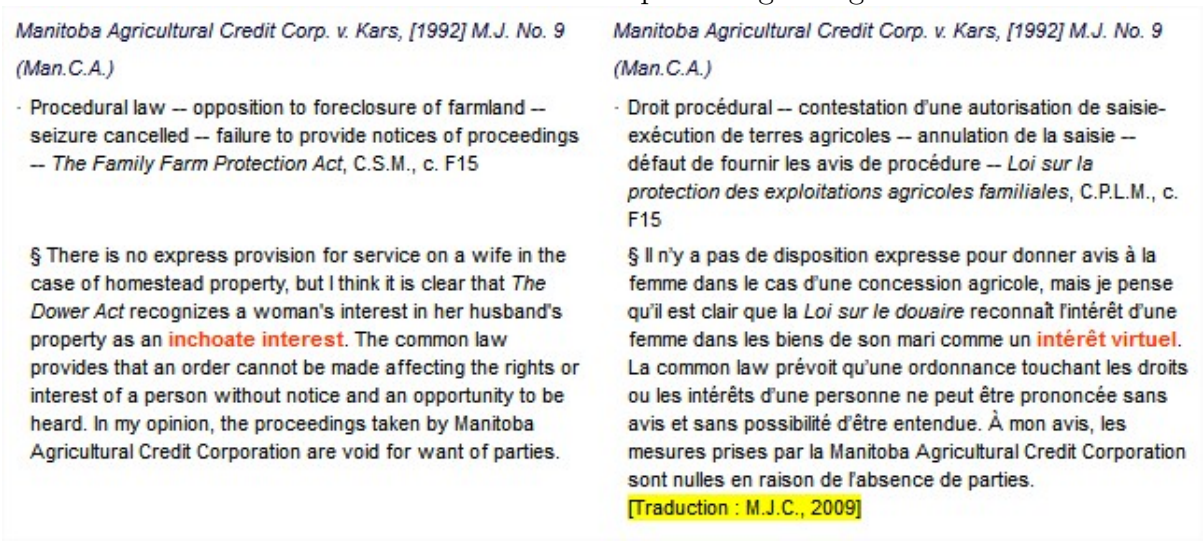
Compte tenu des avantages et des inconvénients de chaque approche, une nouvelle approche est utilisée aujourd'hui : l'approche hybride. Les logiciels actuels [1] de traduction utilisent les forces de chacune des approches afin d'obtenir des résultats efficaces [Kouassi, 2009, Systran, 2012].

5.3 Approche Statistique

Il pourrait paraître surprenant au premier abord de vouloir traiter un processus linguistique comme la traduction par des méthodes statistiques. Toutefois, la traduction d'un texte nécessite la prise de décisions : choisir un mot, une locution ou tournure de phrase en prenant en considération des dépendances souvent difficiles à quantifier. L'approche statistique rend compte de ces dépendances floues et est en mesure de les combiner de façon multiplicative ou additive. De plus, le traitement statistique permet de garantir que pour toute phrase source, une phrase traduite sera générée. En effet, même si la syntaxe de cette phrase n'est pas correcte, elle permettra très probablement de transmettre le sens de la phrase originale. On peut

1. Par exemple : Google Translate, Systran, etc.

FIGURE 5.2 – Corpus bilingue aligné



Manitoba Agricultural Credit Corp. v. Kars, [1992] M.J. No. 9 (Man.C.A.)

- Procedural law -- opposition to foreclosure of farmland -- seizure cancelled -- failure to provide notices of proceedings -- *The Family Farm Protection Act*, C.S.M., c. F15

§ There is no express provision for service on a wife in the case of homestead property, but I think it is clear that *The Dower Act* recognizes a woman's interest in her husband's property as an inchoate interest. The common law provides that an order cannot be made affecting the rights or interest of a person without notice and an opportunity to be heard. In my opinion, the proceedings taken by Manitoba Agricultural Credit Corporation are void for want of parties.

Manitoba Agricultural Credit Corp. v. Kars, [1992] M.J. No. 9 (Man.C.A.)

- Droit procédural -- contestation d'une autorisation de saisie-exécution de terres agricoles -- annulation de la saisie -- défaut de fournir les avis de procédure -- *Loi sur la protection des exploitations agricoles familiales*, C.P.L.M., c. F15

§ Il n'y a pas de disposition expresse pour donner avis à la femme dans le cas d'une concession agricole, mais je pense qu'il est clair que la *Loi sur le douaire* reconnaît l'intérêt d'une femme dans les biens de son mari comme un intérêt virtuel. La common law prévoit qu'une ordonnance touchant les droits ou les intérêts d'une personne ne peut être prononcée sans avis et sans possibilité d'être entendue. À mon avis, les mesures prises par la Manitoba Agricultural Credit Corporation sont nulles en raison de l'absence de parties.
[Traduction : M.J.C., 2009]

donc résumer la traduction statistique comme la combinaison d'une modélisation linguistique et d'une prise de décision statistique [Déchelotte, 2007].

La traduction automatique statistique doit ses origines aux travaux de Brown, eux-mêmes basés sur les modèles de la théorie mathématique de distribution et d'estimation probabiliste de Frederick Jelinek, du MIT [Youssef, 2008].

La traduction par méthodes statistiques repose sur un corpus parallèle, comme la traduction à base d'exemples vue à la section 5.2.4 et plus précisément, un corpus bilingue aligné comme le montre la figure 5.2 [2].

L'alignement a pour objectif de créer un lien au niveau de la phrase entre la phrase dans la langue source et sa traduction dans la langue cible. Notons que l'utilisation des termes "phrase" et "mot" dans l'approche n'a pour but que de faire une référence aux composantes des langues que l'homme utilise durant son apprentissage. En effet, pour les calculs, il ne s'agit que d'une séquence de caractères plus ou moins longue. L'alignement sert donc à limiter le nombre de caractères sur une ligne et que les séquences de caractères limitées sur une ligne dans la langue A soit mises en parallèle avec les séquences de caractères dans la langue B. En considérant les mots, les phrases,. . .comme des séquences de caractères, l'approche se trouve être indépendante des structures de phrases et donc des langues en elles-mêmes. Ensuite, les systèmes statistiques calculent la probabilité qu'une certaine séquence de caractères d'un coté du corpus (Langue A) se répète et ce, en parallèle avec une autre séquence de caractères de l'autre coté du corpus (Langue B).

Par exemple, considérons la phrase suivante "*Il y a un problème*" mise en parallèle [3] dans un corpus avec la phrase "*There is a problem*". Si les deux séquences se répètent un certain nombre

2. Cet extrait est tiré du site web du Dictionnaire juridique de la propriété au Canada à l'adresse `http://www.dualjuridik.org/DUALJURIDIK_WEB/FR/Traduction_Frameset.htm`.

3. Donc, sur la même ligne.

de fois, l'analyse augmente la probabilité que la séquence "*Il y a un problème*" corresponde à la séquence "*There is a problem*". De notre point de vue, l'analyseur estime qu'il y a une énorme probabilité que la phrase "*Il y a un problème*" se traduise par la phrase "*There is a problem*". Ce que nous pouvons estimer comme correcte. Dans la continuité, l'analyse prend aussi en compte l'expression plus petite "*Il y a*", liée à "*There is*". Nous conviendrons que ces expressions sont souvent utilisées dans les textes de leur langue respective. De cette façon, l'analyseur augmente significativement la probabilité que la séquence "*Il y a*" se "traduise" par "*There is*". Petit à petit, l'analyse calcule les probabilités de liaisons [4] pour toutes les séquences du corpus. De cette façon, quand ensuite nous demandons une traduction d'une phrase, autrement dit, d'une séquence ou d'une suite de séquences d'une langue A, le traducteur consulte les probabilités de liaisons avec les séquences dans la langue B et génère la suite des séquences ayant les plus grandes probabilités. Ceci nous donne la traduction. Voila où réside toute la force de l'approche statistique.

Le processus expliqué dans l'exemple ci-dessus s'appelle l'entraînement. C'est durant cette étape que les systèmes statistiques apprennent le modèle probabiliste de traduction. Ils utilisent aussi la partie du corpus contenant les phrases de la langue cible afin de créer un modèle probabiliste de cette dernière. Ces deux modèles sont représentés en pratique par des ensembles de tables contenant les valeurs de probabilités de certains paramètres. C'est à partir de ces deux modèles que le traducteur est ensuite en mesure de générer la traduction la plus probable possible, et ce, sans jamais prendre en considération ne fusse qu'une seule règle de structure d'une des langues. De plus, les probabilités étant calculées pour les liaisons de séquences de la langue A avec des séquences de la langue B, les liaisons n'ayant pas de sens précis, les modèles probabilistes de traduction sont valides pour les deux sens de traduction, sans devoir répéter l'opération.

En général, la qualité des traductions générées par un tel système croît avec la quantité des données sur lesquelles les paramètres du système sont estimés. Autrement dit, l'approche statistique de la traduction automatique est capable de s'améliorer avec des nouvelles données d'entraînement. En effet, plus les corpus sont grands, plus il y a d'informations à traiter, plus proches de la réalité sont les probabilités calculées.

Notons que cette approche, l'approche statistique, est celle principalement utilisée par Google pour son traducteur en ligne, Google Traduction [5]. Profitant en premier lieu de l'ensemble des ressources présentes sur le web, telles que les traductions, Google a mis les moyens nécessaires à l'implémentation de cette approche. Par la suite, l'entreprise a même décidé de numériser des quantités énormes de livres et leurs traductions afin de compléter son corpus. Afin d'encore mieux comprendre cette approche, le lecteur est invité à regarder la vidéo explicative de Google disponible à l'adresse suivante : `http://translate.google.fr/about/intl/fr_ALL`.

4. Et donc de traductions.

5. `www.http://translate.google.com`

Finalement, avertissons le lecteur que ce paragraphe ne constitue en soi qu'une introduction à la théorie de l'approche statistique considérée comme suffisante pour comprendre l'expérience réalisée dans ce mémoire. Cependant, le lecteur curieux pourra consulter les références [Monty, 2010, Youssef, 2008, Haithem, 2010, Monty, 2010] afin d'en savoir plus sur l'approche du point de vue mathématique.

5.4 Outils

Afin de conclure ce chapitre, voyons un peu les moyens qui permettent de mettre en œuvre cette approche. Ainsi, il existe un certain nombre de logiciels de traduction automatique, dont certains d'entre eux sont disponibles directement en ligne. Wikipedia en référence une liste : `http://en.wikipedia.org/wiki/Machine_translation`. Parmi ceux-ci, quelques-uns bien connus tels que Google Traduction, Systran, Bing Translator (Microsoft), etc.

Cependant, ceux-ci ne proposent pas tous la possibilité de créer son propre traducteur, c'est-à-dire, dans le cas de la TA à base de règles, de spécifier ses propres règles linguistiques, ou bien, dans le cas de la TA à base de corpus, de pouvoir entrainer ses propres corpus. Pour trouver ces derniers, il faut évidemment filtrer la liste.

Après comparaison [Wiki, a], il en existe un petit nombre qui peuvent répondre à nos exigences. Nous notons :

- Apertium : `http://www.apertium.org/` - À base de règles.
- Moses : `http://www.statmt.org/moses/` - Approche Statistique.
- Lets'MT ! : `https://www.letsmt.eu/Start.aspx` - Approche Statistique - En ligne.
- etc : `http://www.computing.dcu.ie/~mforcada/fosmt.html`

Certains de ceux-ci proposent aussi l'avantage d'être open source. Autrement dit, leur code est disponible en téléchargement, rendant de ce fait le projet modifiable et adaptable. Très utile dans le domaine de la recherche, vu que ceux-ci permettent alors d'implémenter des théories à l'état de recherche.

Notons que c'est le cas de Moses, l'outil que nous utilisons dans notre expérience. Le choix de prendre Moses s'est fait par défaut, étant donné que celui-ci était déjà disponible dans le laboratoire de recherches.

5.4.1 Moses

Moses est une boîte à outils très performante qui implémente les algorithmes d'apprentissage et de décodage pour les systèmes de traduction automatique statistique. La boite à outils contient un certain nombre de scripts préfaits qui réalisent toutes les étapes de création d'un traducteur automatique, notamment avec les options par défaut : scripts de préparations, d'entrainements, etc. Par exemple, lors de l'entrainement d'un corpus bilingue, un script réalise

l'ensemble des étapes nécessaires à la conception des modèles probabilistes de traduction et de la langue cible, comme expliqué à la section 5.3. Ainsi, en effectuant quelques commandes, on obtient rapidement un traducteur avec lequel on peut réaliser des tests de traductions.

Nous ne détaillons pas l'ensemble des commandes de Moses. En effet, celles utilisées lors de l'expérience sont expliquées en même temps que leur utilisation dans la deuxième partie de ce mémoire. Cependant, le lecteur curieux d'en apprendre plus est invité à lire son manuel [Koehn, 2011] disponible sur le site web de Moses.

De plus, nous ne détaillons pas non plus toute la machinerie d'implémentation de Moses. Le lecteur aussi curieux d'en savoir un peu plus est invité à jeter un œil sur l'article scientifique [Hoang *et al.*, 2007].

Moses est téléchargeable gratuitement et continue d'être développé à l'adresse suivante : `http://www.statmt.org/moses/`

Deuxième partie

Etude de faisabilité

Chapitre 6

Idée de Recherche

L'état de l'art a naturellement apporté toutes les pièces nécessaires à la compréhension des chapitres suivants. Cette deuxième partie vise à expliquer en détail les différentes étapes effectuées dans le cadre de l'étude de faisabilité. Rappelons cette dernière : "Étude de faisabilité de la génération automatique[1] *de commentaires*[2] *à l'aide des techniques de la Traduction Automatique Statistique*[3] *".*

Ce chapitre introduit l'expérimentation et explique brièvement les différentes étapes qui sont développées par la suite. En premier lieu, l'énoncé initial de recherche est cité. Ensuite, nous l'éclaircissons en présentant l'objectif et le principe qui sous-tendent l'idée de recherche. Finalement, nous introduisons brièvement les différentes étapes qui permettent d'atteindre l'objectif, qui est d'évaluer la faisabilité de générer automatiquement des commentaires grâce à la traduction automatique statistique.

6.1 Énoncé

" Le sujet consiste à étudier la possibilité de générer des commentaires pour des éléments de programmes, tels que les méthodes à partir de corpus de données (paires code-commentaire). Pour ce faire, nous pensons à l'utilisation des techniques de traduction statistiques qui utilisent une base d'exemples (paire de segments de texte équivalents de deux langues). Ces techniques de traduction donnent des résultats impressionnants (voir par exemple les nouveaux traducteurs de Google). Le projet consistera dans une première phase à créer un corpus de données (méthodes et leurs commentaires) à partir des logiciels open source. La deuxième phase consiste à adapter des traducteurs existants à la problématique de la génération des commentaires. "

1. Chapitre 4 de l'État de l'art
2. Chapitre 3 de l'État de l'art
3. Chapitre 5 de l'État de l'art

6.2 Objectif et Principe

La question se pose facilement : peut-on envisager de générer des commentaires à l'aide de la traduction automatique statistique ? C'est ce à quoi nous allons essayer de répondre grâce à notre expérience, devenant de ce fait notre objectif.

Mais quel est le principe qui sous-tend notre expérience et l'idée de recherche ? Tel qu'expliqué dans le chapitre sur la traduction automatique statistique, l'approche statistique (section 5.3) vise, à partir d'un corpus bilingue aligné, à calculer les probabilités qu'une séquence de caractères d'un côté du corpus (Langue A) soit liée à une autre séquence de l'autre coté du corpus (Langue B), et ce, par alignements et répétitions. Ainsi, nous avons montré que l'avantage de l'approche vient de l'indépendance vis-à-vis des langues et de leur structure. Cet aspect permet d'imaginer la possibilité que la langue source soit le code source et que la langue cible soit les commentaires. Après entraînement, nous aurons un modèle probabiliste de traduction, qui, utilisé avec un traducteur, serait capable de "traduire" du code source en commentaires. Autrement dit, il serait possible de générer automatiquement des commentaires à partir du code source.

Pour ce faire, nous avons vu à la section 5.4 du chapitre 5 les outils qui nous permettent de réaliser cette approche, soit d'être capable de générer un traducteur à partir d'un corpus de textes. Cependant, avant d'utiliser un outil, il nous faut un corpus, un bitexte et plus exactement un corpus bilingue aligné de code et de commentaires. Ainsi, l'étape première de notre expérience consistera en la conception du corpus. Ensuite, nous l'entraînons à l'aide de l'outil choisi, c'est-à-dire Moses. Cela fait, nous aurons les composants nécessaires afin de tester des traductions et ainsi apporter une réponse à la question de départ. C'est ainsi que se résume notre expérience.

6.3 Structure

La première étape de l'expérimentation, présentée dans le chapitre 7, vise donc à préparer un bitexte de codes et de commentaires. Cette préparation du corpus se réalise en deux phases : la phase d'extraction et la phase de préprocessing.
La première phase a pour objectif d'extraire les lignes de code et de commentaires des fichiers de code source d'un projet de programmation, afin de constituer le corpus. Techniquement, cela a demandé l'implémentation d'outils qui sont expliqués plus en détail dans le chapitre 7. De plus, l'extraction est aussi le moment où l'on choisit ce qui compose le corpus. En effet, toute ligne écrite dans une implémentation (commentaires/lignes de code) n'apporte pas toujours des informations intéressantes en vue de la traduction. Pour cette raison, des hypothèses de sélection de certains éléments de programmes sont déjà réalisées lors de l'extraction.

La deuxième phase, appelée préprocessing, consiste au nettoyage du corpus une fois ce dernier extrait. En effet, quelque soient les hypothèses de sélection d'éléments de programmes explicitées lors de l'extraction, certains éléments, plus petits, utilisés lors de l'activité de programmation, se retrouvent dans le corpus et provoquent du bruit lors de l'entraînement. C'est le cas des caractères spéciaux tels que les parenthèses, les points-virgules, etc. De plus, d'autres éléments tels que les noms de méthodes sont splittés afin de donner plus d'informations lors de l'entraînement. Ainsi, l'activité de préprocessing vise à identifier et déterminer un ensemble d'hypothèses qui, en fonction de leurs états sur le corpus, vont avoir une influence sur la qualité de la génération. Techniquement, des outils ont été implémentés afin de répondre à ces besoins et sont présentés également dans le chapitre 7.

Après l'étape de préparation, le bitexte est considéré comme prêt, i.e. présentable en entrée de l'outil de création de systèmes de traduction automatique statistique. Afin de créer le traducteur de code-commentaires, autrement dit notre générateur de commentaires, il faut entraîner le corpus parallèle à l'aide de Moses. Ceci est la deuxième étape. Celle-ci se matérialise par l'exécution d'une commande bash que nous verrons plus en détail dans le chapitre 8. Ce chapitre présente aussi les différentes configurations de bitextes. Une configuration est une modification du bitexte. Ainsi, l'évolution du corpus au cours de l'expérience et à des fins de tests, ainsi que les raisons de cette évolution, sont décrites dans le chapitre 8.

Grâce à l'étape d'entraînement, nous avons notre traducteur/générateur de code-commentaires. L'évaluation de la qualité des traductions de ce dernier amène à évaluer la faisabilité de l'idée de recherche. Ainsi, en premier lieu, nous nous devons de réaliser des tests de traductions à partir des outputs de l'entraînement. C'est la troisième étape de l'expérimentation et celle-ci est présentée dans le chapitre 9. Dans la continuité de l'étape de traduction, nous décrivons la quatrième et dernière étape de l'expérimentation : l'évaluation. Ainsi, la suite du chapitre présente les différentes évaluations de l'évolution de la traduction réalisées dans le cadre de l'expérimentation. Celles-ci y sont détaillées sous forme de constatations personnelles et chiffrées.

Finalement, le chapitre 10 présente la conclusion en résumant le parcours effectué, le résultat de l'étude et décrit les limites rencontrées durant la recherche et les éventuelles pistes de solutions à envisager en cas de futurs travaux dans le domaine de l'expérience.

Chapitre 7

Préparation du corpus

Tel qu' énoncé dans le chapitre précédent, la première étape de l'étude de faisabilité de la génération automatique des commentaires par l'intermédiaire de la traduction automatique statistique est la préparation d'un corpus parallèle de code et de commentaires. Dans le cadre de notre expérience, il ne s'agit pas seulement de la première étape, mais bien de la plus importante. En effet, vu le temps imparti à cette recherche durant le stage et son objectif initial[1]*, l'évaluation de la qualité de la traduction ne dépend au final que de l'étape de préparation du corpus. Autrement dit, améliorer la qualité de la génération des commentaires en envisageant une modification de l'outil de création de traducteurs automatiques statistique n'a jamais été prévu à court terme. Pour ces raisons, l'étape de préparation et ses deux phases, l'extraction et le préprocessing, sont les plus importantes.*
Ces phases sont expliquées en détail dans ce deuxième chapitre de la deuxième partie du mémoire. Tout d'abord, nous définissons la notion de corpus, afin que le lecteur puisse clairement se faire une représentation de la "matière première" que nous utilisons tout au long de cette recherche. Ensuite, nous expliquons les composants que nous avons appris à maîtriser et ceux que nous avons développés afin de réaliser la phase d'extraction. Finalement, nous détaillons les développements effectués sur bases des hypothèses de nettoyage de corpus choisies dans le cadre du préprocessing. Avant de commencer, rappelons que les détails de l'implémentation des développements sont situés en annexe.

7.1 Définition : Corpus

Commençons ce paragraphe en citant la définition du terme "corpus" :

> Un corpus est un ensemble de documents, artistiques ou non (textes, images, vidéos, etc.), regroupés dans une optique précise. On peut utiliser des corpus dans plusieurs domaines : études littéraires, linguistiques, scientifiques, etc [Wiki, b].

1. Évaluer si l'idée de recherche est réalisable.

Le lecteur attentif aura remarqué l'utilisation courante du terme "bitexte" dans notre texte. Utilisé comme synonyme du mot "corpus", la raison est simple. Dans le cadre de l'utilisation de l'outil Moses, notre corpus se doit de se présenter sous forme d'un corpus parallèle.

> On appelle corpus parallèle un ensemble de couples de textes tel que, pour un couple, un des textes est la traduction de l'autre. Il est intéressant d'aligner ces corpus, c'est-à-dire de faire correspondre chaque unité du texte en langue source avec chaque unité de texte en langue cible (au niveau des paragraphes, phrases et mots) pour disposer d'un jeu de donnée bilingue, en particulier dans des domaines spécialisés où le vocabulaire et l'usage des mots et des expressions évoluent rapidement [Wiki, b].

À titre d'exemple, le lecteur peut consulter la figure 5.2.

Maintenant que le lecteur peut se représenter un corpus parallèle, la suite est simple à imaginer : il nous faut préparer un bitexte, avec d'un coté le code source et de l'autre coté, les commentaires du dit code source (la "traduction"). Pour ce faire, nous allons extraire ces différents éléments de projets de programmation, et exactement dans notre cas, de classes Java.

7.2 Phase d'extraction

L'extraction est la première phase de la préparation du corpus. Elle consiste à extraire des lignes de code et leurs commentaires respectifs des fichiers de code source de projets de programmation et ceci afin de constituer le corpus. Pour réaliser cette phase, il a fallu implémenter des outils, dont le code source est disponible en annexe. Aussi, ces outils ont utilisés des composants que nous présentons ci-dessous. L'implémentation a été réalisée pour des projets codés en Java. Le choix de Java s'est fait par défaut, ce dernier étant celui utilisé majoritairement dans la laboratoire. Cependant, nous rappelons que c'est aussi un des langages les plus utilisés aujourd'hui [Tiobe, 2012].

Rapidement, une classe Java est principalement constituée d'une en-tête de classe, d'en-têtes de méthodes et leurs implémentations et finalement, de commentaires. Ce sont ces éléments de programmes qui nous intéressent dans la conception du corpus. Dans les deux sections qui suivent, nous présentons les composants que nous avons utilisés à la conception des outils permettant de réaliser l'extraction de ces éléments de programmes.

7.2.1 Composants d'implémentation

SableCC

SableCC est premièrement défini comme *un framework orienté-objet qui génère des compilateurs en Java*. Dans notre cas, c'est sa fonction d'analyseur syntaxique qui nous intéresse.

Un analyseur syntaxique (ou parser, en anglais) est un programme qui consiste à mettre en évidence la structure d'un texte, et qui dit texte, dit programme informatique. C'est cette structure qui nous intéresse puisque c'est elle qui va nous permettre de localiser les éléments de programme que nous avons décrit plus haut. Cette structure est retournée par SableCC sous forme d'arbres syntaxiques codés en XML, comme le montre la figure 7.1. C'est en exécutant des requêtes XPATH sur ces arbres syntaxiques que nous localisons et extrayons les éléments de programmes afin de les placer dans le corpus. L'utilisation d'XPATH est détaillée dans le prochain paragraphe.
Le laboratoire dans lequel la recherche a été effectuée ayant déjà implémenté des outils d'extractions de commentaires pour d'autres études, SableCC s'est imposé par défaut.

XPATH et Regex

XPATH est un langage de requêtes, requêtes qui permettent d'identifier un nœud ou un ensemble de nœuds dans un document XML. Rappelons-le, SableCC retourne un arbre syntaxique codé en XML d'un programme, par exemple une classe Java. En d'autres termes, les éléments de programmes dont nous avons besoin sont représentés par des nœuds XML dans le fichier résultant de l'analyse syntaxique. Nous devons les atteindre, parfois exécuter des opérations supplémentaires et finalement les extraire afin de les placer dans un fichier texte qui sera utilisé par Moses lors de l'entrainement. Par définition, XPATH répond parfaitement à nos besoins. Rajoutons cependant l'utilisation en parallèle d'expressions régulières (Regex). Celles-ci permettent d'effectuer un ensemble d'opérations très puissantes dans la comparaison de chaines de caractères. La comparaison de chaines de caractères étant fortement utilisée lors de requêtes XPATH sur des nœuds XML, l'utilisation d'expressions régulières a permis d'une part de simplifier certaines expressions, mais aussi d'en optimiser certaines.

Le lecteur curieux est invité à jeter un œil sur les annexes afin de connaître les détails de l'implémentation des requêtes XPATH ainsi que de l'utilisation des expressions régulières (Regex). De plus, rajoutons que le web regorge de ressources concernant l'utilisation de XPATH.

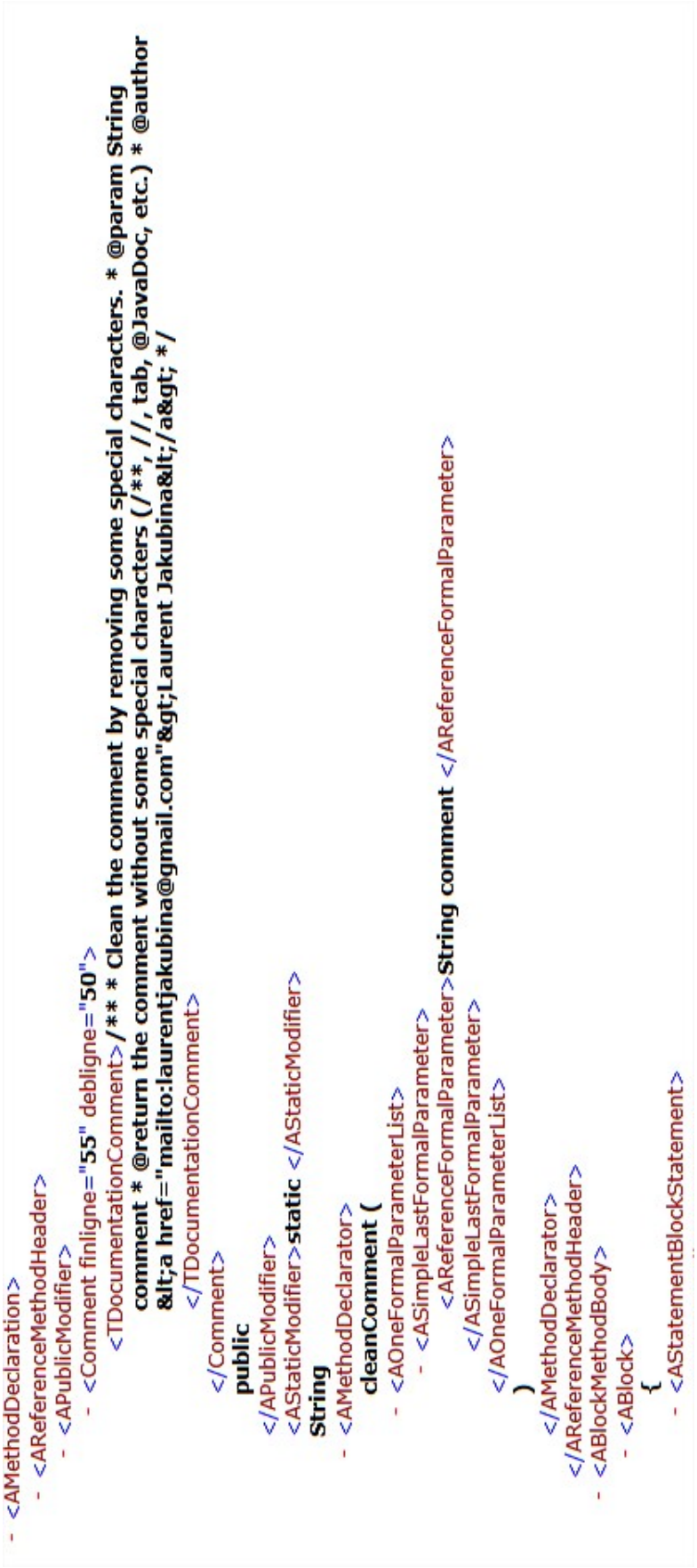
7.2.2 Hypothèses d'extraction

Durant l'implémentation des différentes méthodes permettant l'extraction des nœuds représentants les différents types d'éléments de programme (en-tête de classe, déclarations de méthodes, corps de méthodes, commentaires, etc.), certains choix d'extraire tel ou tel élément ont déjà été effectués. Ils sont présentés ici car ils impliquent indirectement des changements sur le corpus et donc sur la qualité de la traduction, étape finale du processus.

– Trois types de commentaires ont été analysés en fonction de leur position vis-à-vis du code source lié : les commentaires de méthodes, les commentaires de classes et finalement les

FIGURE 7.1 – Extrait d'un arbre syntaxique au format XML.

En regardant attentivement, on remarque les balises qui décrivent la présence d'un commentaire (<Comment >), d'une en-tête de méthode (<AMethodDeclarator >) et le début du corps d'implémentation de la méthode (<ABlock >).



commentaires de code. Après observations, il a été choisi de ne garder que les méthodes avec leurs commentaires respectifs (choix du 20/10/2011). Ainsi, le corpus ne contient pas de code lié à la déclaration de classes Java ni de commentaires présent dans le but de documenter des petits morceaux de code dispersés.

– Un autre choix a été effectué concernant l'extraction des informations qu'une méthode peut retourner. En effet, une méthode donne de l'information par l'intermédiaire de son en-tête mais aussi par l'implémentation dans son corps. Ainsi l'extraction prend l'en-tête de la méthode ainsi que son corps d'implémentation.

En conclusion, le corpus est constitué, pour la partie code, des déclarations ainsi que du corps d'implémentation des méthodes et pour la partie commentaire, des commentaires qui sont écrits pour ces méthodes.

7.3 Phase de préprocessing

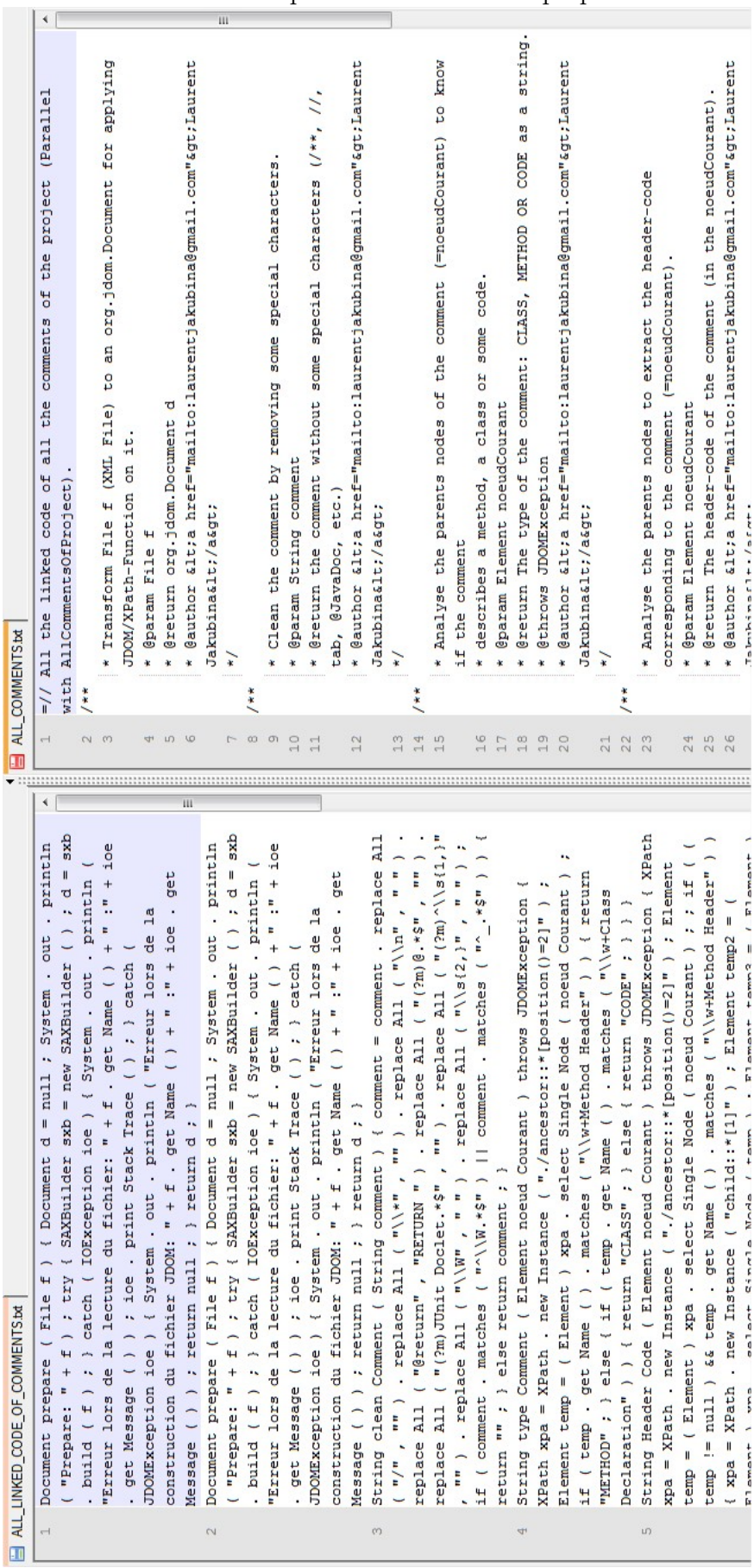
À ce moment-ci du processus, comme le lecteur peut l'observer sur la figure 7.2, le corpus est extrait en ayant respecté les contraintes que nous nous sommes fixées. Cependant, il reste un certains nombres d'éléments qu'il convient d'éliminer tels que les caractères spéciaux (exemple : “.”, “;”, “,”, etc.) ou les mots clés Java. En effet, leur présence dans le corpus implique qu'ils ont nécessairement un impact lors de l'entraînement effectué par le générateur de traducteur Moses, et donc sur la traduction. Encore plus par leur énorme quantité dans le bitexte. Rappelons que l'approche utilisée lors de notre recherche, l'approche statistique, effectue des calculs de probabilités basés sur la répétition d'un certain nombre d'éléments dans le corpus. Ainsi, ces éléments, par leur présence en grande quantité, produisent énormément de bruit lors de l'entraînement. Pourquoi du bruit ? Car il va de soi qu'une virgule au milieu d'une phrase, ou un point à la fin de celle-ci n'est pas ce qui apporte le plus d'informations lors d'une traduction. De ce fait, il s'avère nécessaire de les nettoyer, de les retirer du corpus et ce, avant l'entraînement. C'est ce que nous appelons le préprocessing.

7.3.1 Hypothèses de préprocessing

Le préprocessing ayant été expliqué et justifié, les points qui suivent listent les différents nettoyages qui sont effectués sur le corpus, après son extraction :

– Nettoyage de tous les caractères spéciaux : “.” |“,” |“;” |“?” |“ :” |“/” |“!” |“(” |“)” |“[” |“]” |etc.
– Nettoyage des lignes JavaDocs sauf du tag “@return” et son contenu.
– Nettoyage des caractères tels que les retours-chariots, les retours à la ligne, ...
– Nettoyages particuliers en vue d'adapter le corpus a être parallèle. En effet, rappelons la contrainte du corpus parallèle : les phrases sources et cibles doivent être placées ligne par

FIGURE 7.2 – Corpus extrait mais non préprocessé



ligne, la ligne du texte source devant être en face de la ligne du texte traduit dans les deux fichiers textes.

– Splitter les identifiants (Voir section 7.3.2 pour plus de détails.)

Le lecteur peut jeter un œil sur les détails d'implémentations des traitements effectués durant la phase de préprocessing en allant voir le code source situé en annexe.

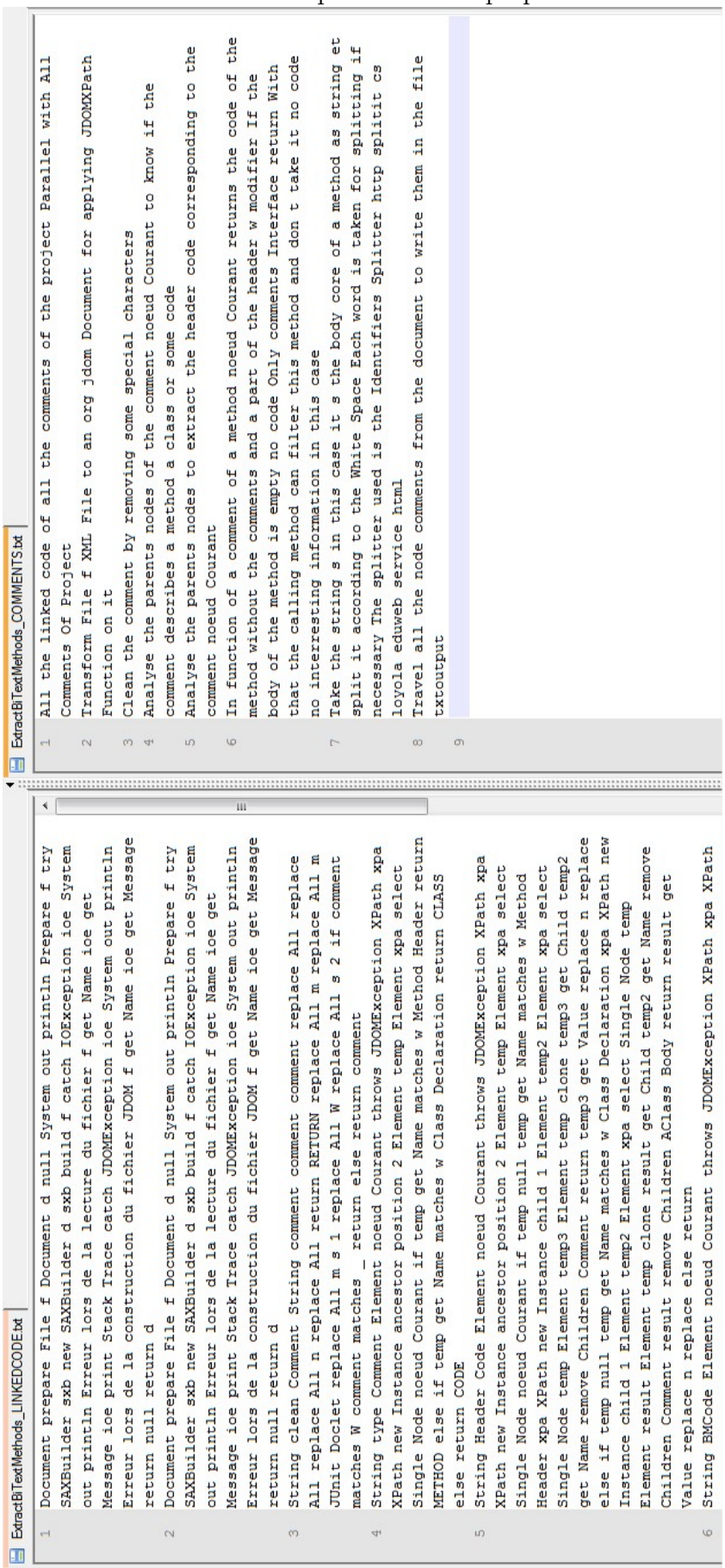
7.3.2 Détails sur le splitteur d'identifiants

Parmi les choix de préprocessing à effectuer sur notre corpus s'est retrouvée l'idée de splitter les identifiants de méthodes. En effet, les noms de méthodes devraient, si les programmeurs respectent les bonnes pratiques, ce que nous admettrons comme généralement vrai pour un programmeur avec un minimum de formation, s'écrire comme ceci : "JeSuisUnBonneMethode" (exemple : SplitIdent()). Un tel nom de méthode est considéré comme une et une seule chaine de caractère, un mot, lors de l'entrainement statistique. Nous admettrons que sous cette forme, ce "mot" n'apporte presque pas ou peu d'informations afin d'effectuer de la traduction. Il apparait en effet plus plausible que le nom de la méthode donne beaucoup plus d'informations utilisables en découpant les différents mots qui composent son nom : "Je Suis Une Bonne Methode". Le nom de la méthode représentant déjà, sous cette forme, une documentation importante de ce qu'elle réalise du point de vue de son implémentation.

Dans cette optique, il a été implémenté ce que nous appellerons un "Splitteur d'identifiants". Son code source est disponible en annexe, dans le fichier ExtractBiTextMethods.Java.

En cette fin de chapitre, le corpus, extrait et préprocessé, est enfin prêt à être entrainé (figure 7.3).

FIGURE 7.3 – Corpus extrait et préprocessé



Chapitre 8

Entraînement du corpus

Grâce au chapitre précédent, nous avons enfin notre corpus parallèle extrait, préprocessé et composé d'un côté de lignes de code et de l'autre, de commentaires. De plus, celui-ci est déjà marqué de quelques hypothèses (qui influence indirectement la traduction et donc la génération des commentaires) que nous avons expliquées en détail (sélection uniquement des méthodes, splitter les identifiants, etc.). D'autres caractéristiques du corpus, par exemple sa taille, ont aussi une influence sur la qualité de la traduction. En effet, plus le corpus est grand, plus il contient d'informations, plus il est capable d'apprendre, meilleures sont les traductions[1]*. C'est le principe de l'approche statistique. Ainsi, l'agrandissement du corpus est aussi une étape qui a son importance pour l'évaluation de la qualité des traductions générées. Pour cette raison, c'est dans ce chapitre que nous détaillerons les différentes versions que le corpus a connues lors des entraînements. Ces différentes versions, nous les appelons "configurations" et elles seront expliquées dans la deuxième partie de ce chapitre.*
La première partie, quant à elle,montre comment entraîner le bitexte. Autrement dit, comme utiliser Moses[2] *: quels sont les fichiers à préparer, quelles sont les commandes à lancer, quels sont les fichiers à récupérer, etc. À la fin de ce chapitre, le lecteur aura les moyens de lancer ses propres entraînements en vue d'effectuer des traductions. Ce chapitre n'a cependant pas la prétention d'expliquer le fonctionnement et la machinerie interne du générateur de traducteurs par l'approche automatique statistique qu'est Moses. Ainsi, le lecteur désireux d'en savoir plus peut, en premier lieu, consulter l'état de l'art et puis, éventuellement, lire le manuel d'utilisation [?].*

8.1 Commandes Moses d'entraînement

L'entraînement du corpus à l'aide de Moses n'a rien de complexe. Il suffit de lancer une série de commandes, en spécifiant certains fichiers en entrée. Principalement, ceux qui représentent

1. C'est aussi le fonctionnement de "Google Traduction", le traducteur en ligne de Google
2. Ce qui suit suppose que l'outil est déjà installé sur la machine du lecteur.

le corpus. C'est-à-dire, deux fichiers textes.

Afin d'illustrer les commandes de manières concrètes, nous allons imaginer l'existence de ces deux fichiers textes : le premier, "memoire.cod", qui contient la partie du corpus de code source. Rappelons qu'il s'agit de notre langue source (qui va être traduite ; qui va "générer" des commentaires). Le deuxième, celui qui représente la langue cible,"memoire.com". Il contient en effet l'ensemble des commentaires extraits.

Avant de lancer la commande principale d'entraînement, Moses conseille d'exécuter une petite série de commandes de nettoyage sur le corpus. En effet, l'étape de préprocessing que nous avons effectuée dans le chapitre précédent n'est pas un pré requis de manière générale à tout corpus. Il est spécifique à notre domaine de recherche, c'est-à-dire le fait de travailler sur du code source et des commentaires. Ainsi, Moses s'assure qu'un certain nombre de caractères spéciaux soient effacés avant l'entraînement. C'est le cas des majuscules (qu'il transforme en minuscules. Cela s'explique aussi par le fait que deux mots identiques, mais avec une majuscule de différence ne sont pas considérés comme un seul et même mot lors de l'entraînement. Le passage en "tout minuscule" évite ce problème) ainsi que des espaces blancs répétitifs.

La première commande est :

```
# clean-corpus-n.perl memoire com cod clean 1 1000000
```

"memoire com cod" est la façon de spécifier l'entrée des deux fichiers "memoire.cod" et "memoire.com". "clean" est le nom que vont porter les deux fichiers de sortie. "1000000" représente le nombre de mots à prendre en compte dans le corpus. Généralement, on prend un très grand nombre.

La deuxième commande de nettoyage est :

```
# lowercase.perl <clean.com>finalclean.com
```

```
# lowercase.perl <clean.cod>finalclean.cod
```

Dans les deux cas, nous spécifions une partie du corpus nettoyé de la commande précédente ("clean.com" — "clean.cod"). Le deuxième paramètre est le nom que l'on va attribuer au fichier en sortie.

La commande suivante est plus importante. Elle génère un fichier particulier, "SRILM_FILE", qui d'une part, doit absolument être mis en paramètre de la commande principale, mais qui d'autre part, contient une représentation de ce qu'on appelle le "modèle de la langue". De la langue cible pour être exact. C'est-à-dire la langue vers laquelle la traduction s'effectue. Pour cette raison, la commande ne doit être lancée que sur le fichier qui représente le corpus de commentaires.

```
# ngram-count -text finalclean.com -lm SRIM_FILE
```

Finalement, la dernière commande, celle qui lance l'entraînement. Attention, l'entraînement peut durer un certain temps en fonction de la taille du corpus et de la machine de calculs. Allant de quelques minutes à quelques heures. Des informations chiffrées plus précises sont données dans la deuxième partie, où sont décrits les différents corpus sur lesquels des entraînements ont été effectués. De plus, en sortie, sont générés un ensemble de fichiers de configurations. Ceux-ci seront utilisés, afin de lancer le traducteur dans le chapitre suivant.

```
# train-model.perl - -corpus finalclean - -root-dir test - -f cod - -e com - -lm
0 :3"..."SRILM_FILE >& training.out
```

Quelques petites informations :

- "finalclean" est le nom commun donné aux deux fichiers du corpus. Spécifié ses deux parties se fait grâce à "- -f cod" (pour le code) et "- -e com" (pour les commentaires).
- "test" spécifie le dossier de destination des fichiers de configurations (- -root-dir).
- "..." est le chemin absolu vers le fichier "SRILM_FILE", le modèle de la langue (- -lm).
- "training.out" est un fichier journal (log) de l'entraînement.

8.2 Configurations du corpus

Comme déjà dit précédemment, *"d'autres caractéristiques du corpus, par exemple sa taille, ont aussi une influence sur la qualité de la traduction. En effet, plus le corpus est grand, plus il contient d'informations, plus il est capable d'apprendre, meilleures sont les traductions. C'est le principe de l'approche statistique."* Ainsi l'expérience a amené à travailler sur différentes versions du corpus, à le voir évoluer et surtout s'agrandir. Ces différentes versions du corpus, appelées "configurations", sont présentées ici. En effet, ces configurations impliquent en premier lieu des conséquences sur l'entraînement, avant d'en avoir sur la traduction. Par exemple, le temps mis par la machine pour entraîner un corpus d'une certaine taille. De plus, ces configurations vont structurer les évaluations (tests de traduction afin d'en évaluer la qualité) qui seront effectuées au prochain chapitre. D'où l'intérêt de préparer le lecteur en les présentant ici.

8.2.1 JHotDraw

JHotDraw [3] est un framework GUI Java open source qui permet de réaliser des graphiques techniques et structurés. Il est présenté par ses auteurs comme avant tout un "exercice de design". En effet, au-delà du développement proprement dit, sa réalisation a d'autres objectifs tels que :

3. http ://www.jhotdraw.org/

FIGURE 8.1 – Ensemble des commandes Moses d'entraînement exécutées

- Familiariser les développeurs avec certains patterns de développement
- Identifier de nouveaux designs patterns et les factoriser
- En faire un exemple d'applications flexibles et bien développées
- ...
- Apprendre et s'amuser

Ainsi, JHotDraw se trouve être un projet bien documenté...et donc satisfaisant pour commencer la constitution de notre corpus. C'est ainsi que JHotDraw a été défini comme la première source de notre corpus [4].

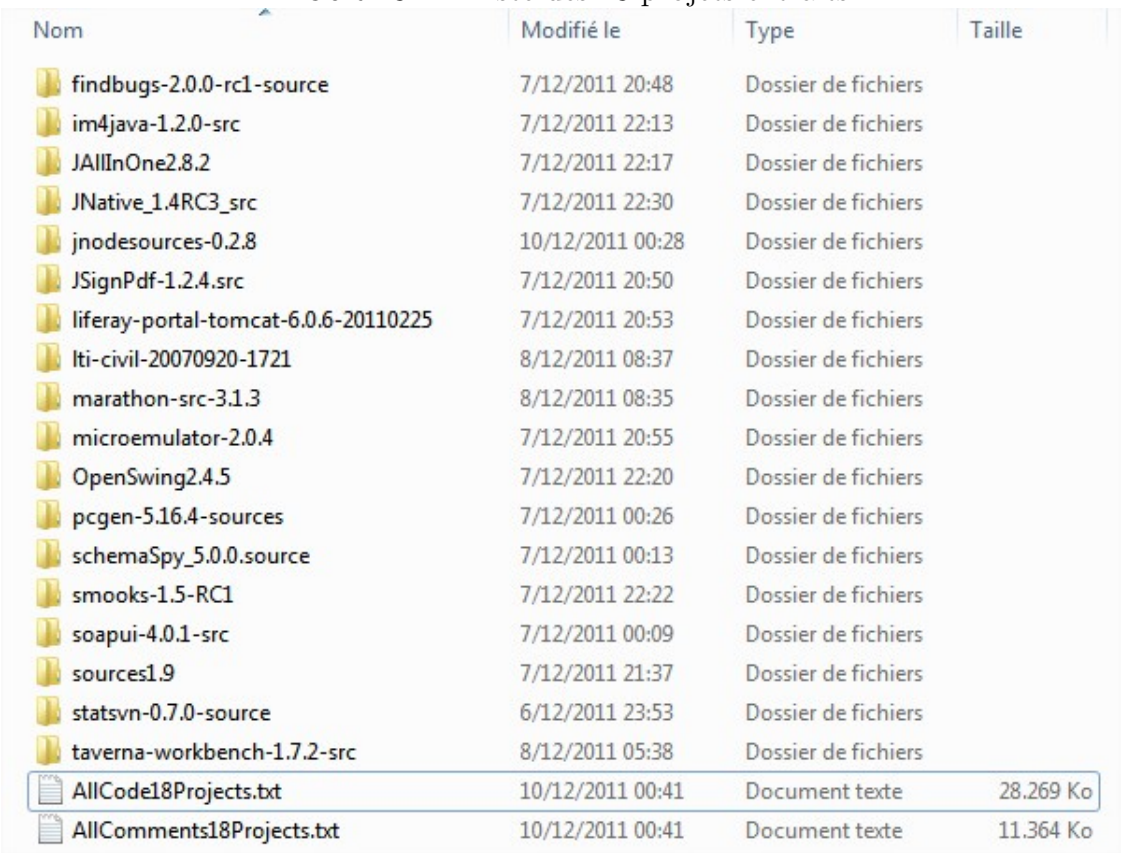
La version téléchargée fut la version 6.0 bêta 1. Après exécution des outils d'extraction sur le projet, le corpus s'est vu composé de 1700 lignes de code-commentaires. Très petit, l'entraînement ne dure qu'une quinzaine de minutes sur une machine moderne. Concernant la qualité des traductions générées à partir de ce corpus, rappelons que les résultats sont présentés dans le prochain chapitre.

8.2.2 JHotDraw - Toutes Versions

Nous savions d'avance que le corpus était trop petit afin de poursuivre l'approche statistique. D'ailleurs, rappelons qu'il a d'abord eu pour vocation de permettre de réaliser les outils d'extraction et de preprocessing. Ainsi, une fois arrivés à l'étape des entraînements de corpus en vue d'évaluer la qualité des traductions et donc des possibilités de générer des commentaires de façon automatique, il fallait agrandir le corpus. Pour cette raison, il a été choisi de constituer le corpus de toutes les versions de JHotDraw disponibles en téléchargement. En plus d'agrandir le corpus, le fait de mettre plusieurs versions d'un même logiciel dans le corpus crée une répétition logique des éléments dans celui-ci. De fait, nous pouvions aussi évaluer les capacités de l'entraînement à réaliser des liaisons codecommentaire sur des éléments très répétés, modifiant leurs probabilités d'apparitions, ce qui conditionnent l'approche statistique. Conclusion : le corpus s'est retrouvé constitué de méthodes et leurs commentaires des versions 5.2 à 7.6 de JHotDraw, soit environ 15000 lignes. Un entraînement de cette version du corpus sur une machine moderne prend environ une heure. À des fins d'évaluations, une version légèrement modifiée de ce corpus a vu le jour. Le but ? Générer des commentaires pour des méthodes de la dernière version de JHotDraw, à partir d'un générateur (donc traducteur de Moses) ayant été entrainé sur un corpus qui contenait toutes les versions de JHotDraw, sauf la dernière. Ce corpus faisait 10500 lignes.

4. Il était d'ailleurs déjà présent lors des tests d'implémentations des outils d'extractions et de preprocessing.

FIGURE 8.2 – Liste des 18 projets extraits



8.2.3 Multiples Projets

La dernière version du corpus avait pour objectif d'agrandir significativement ce dernier. Afin de réaliser celui-ci, 18 projets open source ont été téléchargés et extraits. Le corpus s'est vu constitué de 100000 lignes. Les derniers tests de traductions ont été effectués sur ce corpus. Les résultats sont présentés dans le prochain chapitre. Concernant le temps d'entraînement, c'est ici que des informations importantes sont à prendre en compte. En effet, l'entraînement n'a jamais pu être terminé sur une machine personnelle moderne. De ce fait, l'entraînement a été réalisé à l'aide de la machine de calculs disponible au laboratoire de la recherche. Sur celle-ci, l'entraînement a duré deux heures.

Chapitre 9

Traduction et Évaluation

Les chapitres précédents ont expliqué comment réaliser un corpus extrait et préprocessé ainsi que la méthode pour lancer un entraînement statistique sur ce dernier. Nous sommes aux dernières étapes du processus : réaliser des traductions et les évaluer. Ou, en d'autres termes, déterminer s'il est possible ou non de générer des commentaires pour des éléments de programmes de manière automatique par la traduction automatique statistique qui est notre objectif de recherche.
La première partie de ce chapitre vise à montrer comment effectuer une traduction : la commande à réaliser, les informations à mettre en paramètres, etc. À l'aide de cette section, le lecteur aura la possibilité de reproduire des traductions, avec le prérequis évident d'avoir réalisé l'entraînement nécessaire, tel qu'exploré dans le chapitre précédent.
Ensuite, nous verrons que même si une traduction, autrement dit une génération de commentaires, est possible, cela n'implique pas nécessairement que les commentaires soient lisibles et de bonne qualité. Or, évaluer les possibilités de générer des commentaires requiert que ceux-ci soient utiles et donc de bonnes qualités. Afin d'apporter une réponse à cette évaluation et valider celle-ci, nous définissons et réalisons des évaluations de traduction que nous présentons dans la deuxième partie de ce chapitre.

9.1 Commande Moses de traduction

Réaliser une traduction est une étape très simple. Il suffit d'appeler la commande *Moses* en spécifiant un certain nombre de paramètres. Parmi ceux-ci :

- Le texte à traduire
- et le fichier de configuration "Moses.ini". Celui-ci est situé dans un des sous-dossier du dossier généré par l'entraînement. Le dossier généré était nommé "test" dans le chapitre précédent. Les "..." représente le chemin absolu à spécifier dans la commande.

echo *'texte_à_traduire'* |Moses -f "..."Moses.ini

FIGURE 9.1 – Exemple d'appel de la commande de traduction Moses

```
root@laurent-laptop:/home/laurent/Bureau/StageUdeMTrainingMoses/5_Corpus18Projects/test/mode
root@laurent-laptop:/home/laurent/Bureau/StageUdeMTrainingMoses/5_Corpus18Projects/test/model
# echo 'boolean is Usable return is Enabled my Is Usable' | moses -f moses.ini
Defined parameters (per moses.ini or switch):
        config: moses.ini
        distortion-limit: 6
        input-factors: 0
        lmodel-file: 0 0 3 /home/laurent/Bureau/StageUdeMTrainingMoses/5_Corpus18Projects/SRI
LM_FILE
        mapping: 0 T 0
        ttable-file: 0 0 0 5 /home/laurent/Bureau/StageUdeMTrainingMoses/5_Corpus18Projects/t
est/model/phrase-table.gz
        ttable-limit: 20
        weight-d: 0.6
        weight-l: 0.5000
        weight-t: 0.20 0.20 0.20 0.20 0.20
        weight-w: -1
Loading lexical distortion models...have 0 models
Start loading LanguageModel /home/laurent/Bureau/StageUdeMTrainingMoses/5_Corpus18Projects/SR
ILM_FILE : [0.000] seconds
Finished loading LanguageModels : [1.000] seconds
Start loading PhraseTable /home/laurent/Bureau/StageUdeMTrainingMoses/5_Corpus18Projects/test
/model/phrase-table.gz : [1.000] seconds
filePath: /home/laurent/Bureau/StageUdeMTrainingMoses/5_Corpus18Projects/test/model/phrase-ta
ble.gz
Finished loading phrase tables : [1.000] seconds
Start loading phrase table from /home/laurent/Bureau/StageUdeMTrainingMoses/5_Corpus18Project
s/test/model/phrase-table.gz : [1.000] seconds
Finished loading phrase tables : [3.000] seconds
IO from STDOUT/STDIN
Created input-output object : [3.000] seconds
Translating: boolean is Usable return is Enabled my Is Usable

Collecting options took 0.000 seconds
Search took 0.120 seconds
returns true if the Usable is Enabled for this Is Usable
BEST TRANSLATION: returns true if the Usable|UNK|UNK|UNK is Enabled|UNK|UNK|UNK for this Is|U
NK|UNK|UNK Usable|UNK|UNK|UNK [111111111]  [total=-607.378] <<0.000, -11.000, -400.000, -419.
290, -8.736, -14.685, -6.297, -16.948, 3.000>>
Translation took 0.120 seconds
Finished translating
root@laurent-laptop:/home/laurent/Bureau/StageUdeMTrainingMoses/5_Corpus18Projects/test/model#
```

Un exemple est présenté à l'aide de la figure 9.1. Entouré en rouge, la commande de traduction ainsi que le résultat de la traduction.

Le lecteur avisé aura sans doute remarqué que malgré sa simplicité, la commande oblige de traduire ligne par ligne. Ceci est évidemment très inefficace si l'on souhaite effectuer une traduction sur un ensemble de lignes (dans la cadre d'une évaluation par exemple). Afin de pallier à ce "problème", un petit outil a été implémenté en Java. Celui-ci permet de spécifier en entrée un fichier contenant un ensemble de lignes à traduire. En sortie, l'outil génère un fichier texte contenant l'ensemble des phrases traduites. Le code source de cet outil est disponible en annexe : GenEchanTrad.java.

Ainsi soit-il, le lecteur est maintenant capable de réaliser des traductions, où plus exactement de générer des "commentaires" de code de façon automatique grâce à la traduction automatique statistique. Pourquoi commentaires est-il entre guillemets ? C'est ce que nous allons voir maintenant, en évaluant la qualité des commentaires générés.

9.2 Évaluation des traductions

L'expérience touche à sa fin[1]. Dans le but de valider cette dernière, il est nécessaire de l'évaluer. L'évaluation se déroule en deux temps. En premier lieu, et comme définis par la structure de la section sur les configurations de corpus du chapitre précédent, nous émettons une appréciation personnelle des résultats de traductions à chaque étape de l'évolution du corpus. Ensuite, nous détaillons une évaluation qui a été réalisée sur la dernière version du corpus, et ce dans le but d'avoir une évaluation chiffrée permettant d'appuyer les résultats.

9.2.1 Évolution du corpus

La première version du corpus était constituée de quelques 1700 lignes de code et de commentaire. Voici quatre tests de traductions, présentant le code à traduire et la traduction — commentaire généré(e) :

Code Source	Commentaire généré
boolean is Usable return is Enabled my Is Usable	True if the Usable and is Enabled the Is Usable
Drawing Editor get Drawing Editor return my Drawing Editor	class to be tested the class to be tested Gets the class to be tested
Align Command create Instance throws Exception return new Align Command Align Command Alignment TOPS get Drawing Editor	Align Command Factory method for instances of the Align Command Align Command Alignment TOPS the class to be tested
void add Handles Figure f List handles add Corner Handles f handles handles add south f handles add north f handles add east f handles add west f	Handles is a figure This the handles a each corner Handles the handles handles a south the handles a north the handles a east the handles a west the

Comme relativement attendus, les résultats sont médiocres. L'énorme présence d'associations de segments code - commentaire erronés (ex : yet - the) montre la nécessité première d'agrandir le corpus afin de fournir plus de données à l'entrainement de Moses.

La deuxième version du corpus faisait 15000 lignes, comprenant d'énormes répétitions de code et de commentaires dû à la composition par le biais de toutes les versions disponibles de JHotDraw.

1. Cfr Portal :)

Code Source	Commentaire généré
boolean is Usable return is Enabled my Is Usable	boolean is Usable return is Enabled my Is Usable
Drawing Editor get Drawing Editor return my Drawing Editor	Drawing Editor get Drawing Editor return Drawing my Editor
Align Command create Instance throws Exception return new Align Command Align Command Alignment TOPS get Drawing Editor	Align Command create Instance throws Exception return new Align Command Align Command Alignment TOPS get Drawing Editor
void add Handles Figure f List handles add Corner Handles f handles handles add south f handles add north f handles add east f handles add west f	Handles void add Figure f List handles Corner add Handles f handles handles add south f handles add north f handles add east f handles add west f

Les résultats sont encore insatisfaisants. Sauf pour un cas, mais tellement trop restreint que l'on ne peut le considérer comme viable [2]. Dû à l'énorme répétition de certaines lignes de code et de commentaires dans le corpus, l'entraînement a enregistré ces lignes d'un seul tenant. En conséquence, lors de certaines traductions, pour une ligne de code particulière, la génération est capable de sortir le commentaire exactement associé. En d'autres termes, ces traductions ne sont valides que pour générer des commentaires à partir du code source de JHotDraw, ce qui n'est évidemment pas le but recherché.

La dernière et troisième version du corpus avait pour objectif d'augmenter significativement sa taille. Le téléchargement des 18 projets open source nous a donné un corpus d'environ 100000 lignes. Malheureusement, les résultats sont restés très insatisfaisants, comme peuvent le montrer les exemples qui suivent.

2. Cas qui est représenté dans le tableau car les lignes de code sélectionnées proviennent du corpus.

Code Source	Commentaire généré
boolean is Usable return is Enabled my Is Usable	returns true if the Usable is Enabled for this Is Usable
Drawing Editor get Drawing Editor return my Drawing Editor	Drawing Editor the Drawing Editor gets the current Drawing Editor
Align Command create Instance throws Exception return new Align Command Align Command Alignment TOPS get Drawing Editor	Align Command a Instance lt p gt Exception the Align Command Align Command Alignment TOPS the Drawing Editor
void add Handles Figure f List handles add Corner Handles f handles handles add south f handles add north f handles add east f handles add west f	add Handles Figure of the List figure to the Corner Handles of the figure to the given vector with handles at the current to the model with the figure to the east and west of the to string

Les commentaires générés restent dans la plupart des cas, des suites de mots insensés, proche d'une génération aléatoire. D'où l'utilisation de guillemets autour du mot "commentaire" au début de cette section. Ainsi, même si génération il y a, ce n'est pas vraiment une génération de commentaires valides. L'évaluation chiffrée qui suit va appuyer ces dires.

9.2.2 Évaluation chiffrée

Afin d'appuyer la validation et surtout les constatations relativement subjectives de la section précédente, une évaluation plus formelle a été définie et réalisée à partir de la dernière version du corpus. Celle-ci est présentée ci-dessous.

Définition

L'évaluation consiste à sélectionner 100 lignes dans le corpus préparé. Donc, en d'autres termes, 100 méthodes ainsi que leur commentaire associé. On garde les commentaires dans un fichier séparé et nous faisons passer les 100 lignes de code dans le traducteur généré [3]. Nous obtenons de cette façon 100 "commentaires" générés, que nous allons pouvoir comparer avec les commentaires originaux.

À des fins d'objectivité pour la comparaison, nous définissons une métrique : si le commentaire généré est capable de transmettre le sens du commentaire original, nous attribuons une valeur de "1". Si le commentaire généré ne permet pas du tout de retrouver le sens du

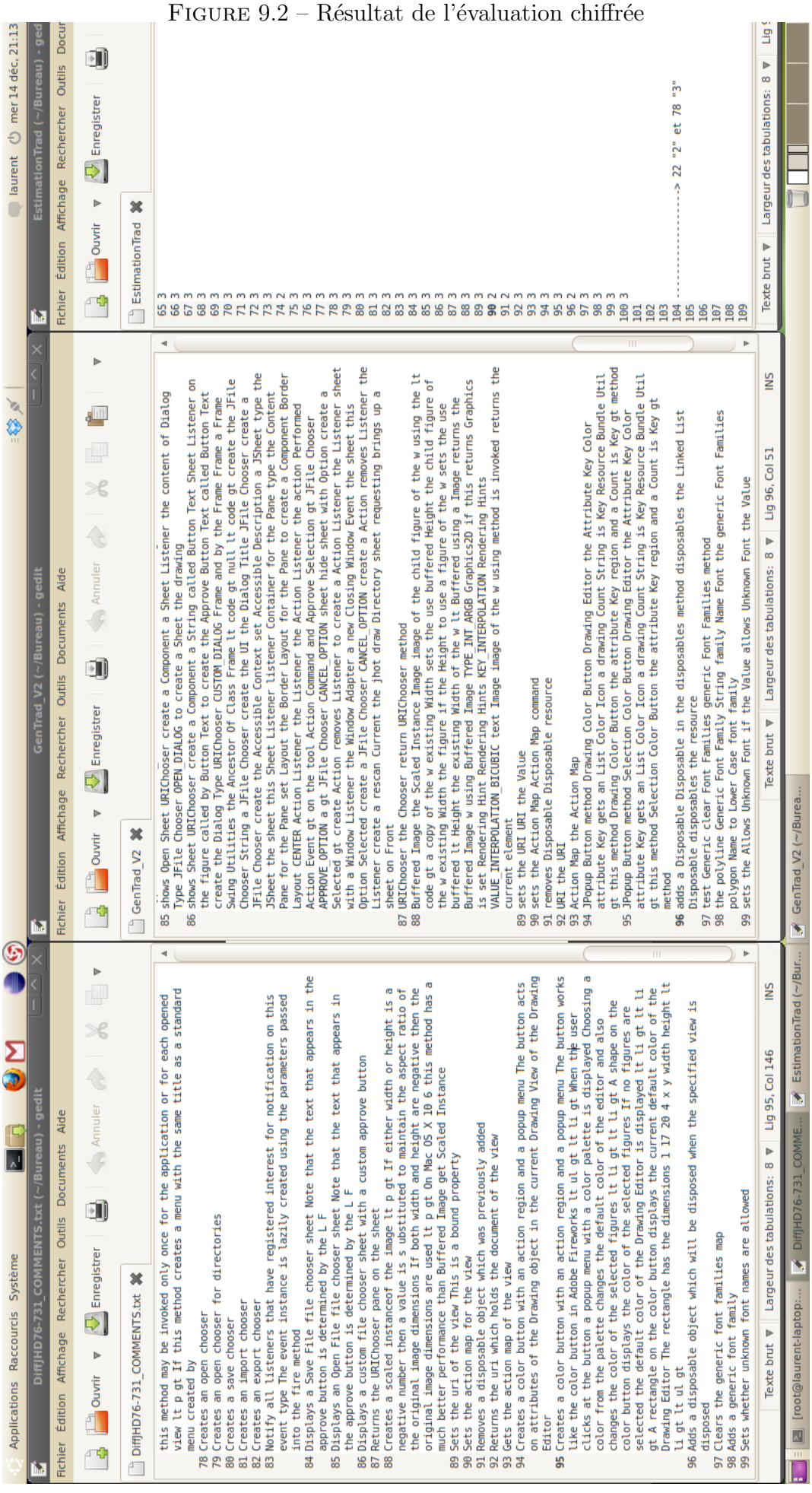
3. Notamment, c'est à ce moment que nous utilisons le petit outil développé en vue de traduire plusieurs lignes à la fois : GenEchanTrad.java dans les annexes.

commentaire original, nous attribuons une valeur de “3”. La valeur intermédiaire “2” émet un avis mitigé.

Résultats

Comme on peut l’observer sur la figure 9.2, il y a 78 “3” soit 78 “commentaires” générés qui ne permettent pas du tout de retrouver le sens du commentaire original. En fait, si l’on essaye de lire entièrement les commentaires générés, on se rend compte que ceux-ci n’ont vraiment aucun sens et ne sont composé que d’une suite de mots, presque placés aléatoirement. De plus, on comprendra que la taille des commentaires vient directement de la taille du corps de la méthode. Les 22 autres commentaires de cote “2” sont des commentaires mitigés, dont on ne peut rien tirer. En conclusion, les résultats sont médiocres.

FIGURE 9.2 – Résultat de l'évaluation chiffrée



Chapitre 10

Conclusion et Travaux Futurs

Dernier chapitre de ce mémoire, "Conclusion et Travaux Futurs" conclut ce dernier. En premier lieu, nous résumons le parcours effectué jusqu'ici ainsi que nos contributions. Ensuite, nous décrivons le résultat de l'étude, en répondant à la question initialement posée dans le chapitre 6 "Idée de Recherche". Et finalement, nous explicitons les éventuelles limites que nous avons pu observer tout au long du développement, tout en essayant de les repousser, ce qui constitue en soi une série de possibilités pour des travaux futurs, si l'idée de "générer des commentaires pour des éléments de programmes à l'aide de la traduction automatique statistique" est poursuivie dans le domaine de la recherche.

10.1 Parcours effectué

À partir de l'énoncé de recherche cité au chapitre 6, nous avons structuré une expérience visant à évaluer la faisabilité de générer des commentaires grâce à l'approche statistique de la traduction automatique. Les outils existants, afin de réaliser concrètement cette approche nous ont permis de cibler les composants que nous devions développer. Ainsi, la première étape consistait à concevoir un bitexte de code et de commentaires. Pour se faire, nous avons développé un ensemble de méthodes [1] permettant d'extraire et de préprocesser le code source et les commentaires. Cette étape est expliquée en détail dans le chapitre 7. À partir de ce corpus, nous étions capables de réaliser un entraînement statistique grâce à l'outil de génération de traducteurs Moses. Cette étape est réalisée dans le chapitre 8 et nous prépare à tester un traducteur de code-commentaire, soit le générateur automatique à évaluer. Cette dernière étape est présentée dans le chapitre 9 où nous montrons comment réaliser des traductions à l'aide de l'outil Moses et des modèles statistiques générés par l'entraînement. Finalement, nous avons défini un protocole d'évaluation sur les traductions générées, permettant d'apporter les résultats à notre étude et conclure celle-ci.

1. Code source disponible en annexes : ExtractBiTextMethods.java

10.2 Résultat

Cette section sera brève, et pour cause : le chapitre précédent a déjà montré les résultats des tests de génération automatique de commentaires par la traduction automatique statistique. Nous avons pu voir que les résultats obtenus sont loin d'être satisfaisants, rendant impossible de confirmer ou d'infirmer la concevabilité de la génération de commentaires d'éléments de programme par la traduction automatique statistique. Cependant, nous nous devons de relativiser. En effet, le corpus utilisé, même lors de la troisième itération avec ces 100000 lignes, reste très petit. En effet, Google Traduction [2], que nous avons souvent cité dans notre texte (et pour cause, rappelons que ce service utilise l'approche statistique dans sa réalisation) est constitué de corpus de quelques milliards de lignes [3]. De ce fait, il pourrait être intéressant d'approfondir l'idée présentée dans ce mémoire en agrandissent le corpus et d'en étudier les nouveaux résultats obtenus.

10.3 Limites et Travaux Futurs

La première limite observée, déjà expliquée dans le paragraphe précédent, concerne la taille du corpus. En effet, celui-ci est trop petit. Une des premières solutions à explorer, afin d'éventuellement améliorer les résultats de l'expérience, serait d'agrandir le corpus de façon significative. Nous parlons d'une centaine de millions de lignes dans le corpus.

Une autre solution viserait à traiter les éléments particuliers des langages de programmation tels que les mots-clés réservés et les identifiants. Rappelons qu'à l'origine, l'application de la traduction automatique statistique s'effectue sur un corpus contenant deux langues naturelles. Ici, nous la réalisons d'une part, sur une langue naturelle [4], et d'autre part, sur un langage de programmation, qui possède des propriétés de syntaxe et de sémantique bien différentes des langages naturels. Ainsi, il serait sûrement très intéressant de prendre en compte ces particularités, notamment lors de l'étape de préprocessing, afin d'améliorer les résultats.

Une troisième possibilité serait d'ignorer, lors de la conception du corpus, certains termes à sémantique trop faible. Tel est le cas des mots grammaticaux comme “le”, “la”, “avec”, ...qui n'apportent pas ou peu d'information à l'entraînement et qui sont omniprésents. Afin d'appuyer cette idée, notons que c'est ce qu'effectue Google lorsqu'un utilisateur effectue des requêtes sur le moteur de recherche [Google, 2012].

La prochaine idée délaisse la modification du corpus comme moyen d'améliorer la performance de la traduction et opte plutôt pour directement modifier l'outil de génération de traducteurs, Moses. En effet, dans son implémention, celui-ci utilise des modèles tels que des

2. Le service de traduction en ligne gratuit de Google.
3. `http://translate.google.fr/about/intl/fr_ALL`
4. Les commentaires.

modèles de langues, . . .qui ont été réfléchis dans le cadre des langues naturelles. De ce fait, nous pourrions envisager de modifier son implémentation afin de prendre en compte les spécificités des langages de programmation. Notons que c'est déjà le cas avec certaines langues telles que le chinois ou l'arabe[Youssef, 2008, Haithem, 2010].

Bibliographie

[piE, 2012] (2012). Les enjeux de l'écrit. http://300gp.ovh.net/~sitecoll/gpi3/site.php?rubrique=71http://300gp.ovh.net/~sitecoll/gpi3/site.php?rubrique=71. [En ligne ; accédé 30-Août-2012].

[Danlos,] Danlos, L. Chapitre 12 : Génération automatique de textes. http://www.linguist.univ-paris-diderot.fr/~danlos/Dossier%20publis/GAT%2700.pdf.

[Déchelotte, 2007] Déchelotte, D. (2007). *Traduction automatique de la parole par méthodes statistiques*. Thèse de doctorat, Université Paris-Sud 11.

[DELORT,] DELORT, J.-Y. Génération automatique de résumés. http://jydelort.appspot.com/resources/papers/techniquesIngenieur07.pdf.

[Deville, 2012] Deville, G. (2012). Info m463 - introduction aux méthodes et concepts d'ingénierie linguistique.

[Fournier, 2012a] Fournier, J.-P. (2012a). Crise du logiciel. http://www.infeig.unige.ch/support/se/lect/gl/gl/node2.html. [En ligne ; accédé 30-Août-2012].

[Fournier, 2012b] Fournier, J.-P. (2012b). Naissance du génie logiciell. http://www.infeig.unige.ch/support/se/lect/gl/gl/node1.html. [En ligne ; accédé 30-Août-2012].

[F.Villeneuve, 2001] F.Villeneuve (2001). Gestion de la documentation des projets informatiques. http://www.dsi.cnrs.fr/conduite-projet/phasedeveloppement/qualite/gestion-doc/guide-gestion-doc.pdf.

[Google, 2012] Google (2012). Recherches google – principes de base. http://www.google.fr/intl/fr/help/basics.html#stopwords. [En ligne ; accédé 30-Août-2012].

[Goudet, 2012] Goudet, J.-L. (2012). Traduction automatique : Les années où tout a changé. http://www.futura-sciences.com/fr/doc/t/informatique-2/d/traduction-automatique-les-annees-ou-tout-a-change_831/c3/221/p2/. [En ligne ; accédé 30-Août-2012].

[Habra, 2010] Habra, N. (2010). Info m110 - ingénierie du logiciel.

[Haiduc *et al.*, 2010] Haiduc, S., Aponte, J., Moreno, L. et Marcus, A. (2010). On the use of automated text summarization techniques for summarizing source code. *In Proceedings of the 2010 17th Working Conference on Reverse Engineering*, WCRE '10, pages 35–44, Washington, DC, USA. IEEE Computer Society.

[Haithem, 2010] Haithem, A. (2010). Approche mixte pour la traduction automatique statistique. Mémoire de D.E.A., Université Stendhal Grenoble 3.

[Hoang *et al.*, 2007] Hoang, H., Birch, A., Callison-burch, C., Zens, R., Aachen, R., Constantin, A., Federico, M., Bertoldi, N., Dyer, C., Cowan, B., Shen, W., Moran, C. et Bojar, O. (2007). Moses : Open source toolkit for statistical machine translation. pages 177–180.

[Hugues, 2002] Hugues, A.-M. (2002). La problematique du logiciel. `http://users.polytech.unice.fr/~hugues/GL/chapitre1.pdf`.

[Hutchins, 1995] Hutchins, W. J. (1995). Machine translation : A brief history. *In CONCISE HISTORY OF THE LANGUAGE SCIENCES : FROM THE SUMERIANS TO THE COGNITIVISTS, PERGAMON*, pages 431–445. Press.

[Jibriss, 2012] Jibriss (2012). L'art de commenter son code. `http://www.jibriss.com/post/2011/09/24/L-art-de-commenter-son-code`. [En ligne ; accédé 30-Août-2012].

[Koehn, 2011] Koehn, P. (2011). *Moses - User Manual and Code Guide*.

[Kouassi, 2009] Kouassi, R. R. (2009). La problématique de la traduction automatique. `http://www.ltml.ci/files/articles4/article_traduction_automatique.pdf`.

[Mati, 2012] Mati, D. (2012). L'histoire des logiciels. `http://membres.multimania.fr/djmati/informatique/Histoire_inform/h_i_logic.htm`. [En ligne ; accédé 30-Août-2012].

[Mireille,] Mireille, C. Cours - la crise du logiciel. `http://www2.lifl.fr/~clerbout/coursIPINT/Cours2-CriseDuLogiciel.pdf`.

[Monty, 2010] Monty, P. P. (2010). Traduction statistique par recherche locale. Mémoire de D.E.A., Université de Montréal.

[N.Rousse, 2004] N.Rousse (2004). Documentation associée à un projet logiciel. `http://www.modelia.org/html/9_fichesTechniques/0080dossierDocLog.pdf`.

[PONTON, 1997] PONTON, C. (1997). Génération automatique de textes : 30 ans de réalisation. `http://w3.u-grenoble3.fr/lidilem/labo/file/GAT97.pdf`.

[Rastkar *et al.*, 2011] Rastkar, S., Murphy, G. C. et Bradley, A. W. J. (2011). Generating natural language summaries for crosscutting source code concerns. *In Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 103–112, Washington, DC, USA. IEEE Computer Society.

[Sridhara *et al.*, 2010] Sridhara, G., Hill, E., Muppaneni, D., Pollock, L. et Vijay-Shanker, K. (2010). Towards automatically generating summary comments for java methods. *In Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 43–52, New York, NY, USA. ACM.

[Strohmeier, 2012] Strohmeier, A. (2012). Cycle de vie du logiciel. `http://diwww.epfl.ch/researchlgl/teaching/case_tools00/doc/cycle-de-vie.pdf`.

[Systran, 2012] SYSTRAN (2012). Traduction automatique : Qu'est ce que c'est ? `http://www.systran.fr/systran/entreprise/technologie/traduction-automatique`. [En ligne ; accédé 30-Août-2012].

[Tiobe, 2012] TIOBE (2012). Tiobe - indicator of the popularity of programming languages. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`. [En ligne ; accédé 30-Août-2012].

[Wiki, a] WIKI. Comparison of machine translation applications. `http://en.wikipedia.org/wiki/Comparison_of_machine_translation_applications`. [En ligne ; accédé 30-Août-2012].

[Wiki, b] WIKI. Corpus. `http://fr.wikipedia.org/wiki/Corpus`. [En ligne ; accédé 30-Août-2012].

[Wiki, 2012a] WIKI (2012a). Bug de l'an 2000. `http://fr.wikipedia.org/wiki/Passage_informatique_%C3%A0_l%27an_2000`. [En ligne ; accédé 30-Août-2012].

[Wiki, 2012b] WIKI (2012b). Commentaire (informatique). `http://fr.wikipedia.org/wiki/Commentaire_%28informatique%29`. [En ligne ; accédé 30-Août-2012].

[Wiki, 2012c] WIKI (2012c). Documentation logicielle. `http://fr.wikipedia.org/wiki/Documentation_logicielle`. [En ligne ; accédé 30-Août-2012].

[Wiki, 2012d] WIKI (2012d). Etude de faisabilité. `http://fr.wikipedia.org/wiki/%C3%89tude_de_faisabilit%C3%A9`. [En ligne ; accédé 30-Août-2012].

[Wiki, 2012e] WIKI (2012e). Générateur de documentation. `http://fr.wikipedia.org/wiki/G%C3%A9n%C3%A9rateur_de_documentation`. [En ligne ; accédé 30-Août-2012].

[Wiki, 2012f] WIKI (2012f). Histoire de l'informatique de gestion. `http://fr.wikipedia.org/wiki/Histoire_de_l%27informatique_de_gestion`. [En ligne ; accédé 30-Août-2012].

[Wiki, 2012g] WIKI (2012g). Traitement automatique du langage. `http://fr.wikipedia.org/wiki/Traitement_automatique_des_langues`. [En ligne ; accédé 30-Août-2012].

[Wiki, 2012h] WIKI (2012h). Vol d'ariane 5. `http://fr.wikipedia.org/wiki/Vol_501_d%27Ariane_5`. [En ligne ; accédé 30-Août-2012].

[Youssef, 2008] YOUSSEF, A. B. (2008). Méthodes mixtes pour la traduction automatique statistique. Mémoire de D.E.A., Université Stendhal Grenoble 3.

Troisième partie

Annexes

Code Source : ExtractBiTextMethods.java

```
1
2  import java.io.File;
3  import java.io.FileNotFoundException;
4  import java.io.IOException;
5  import java.io.PrintWriter;
6  import java.util.Arrays;
7  import java.util.Iterator;
8  import java.util.List;
9  import java.util.regex.Matcher;
10 import java.util.regex.Pattern;
11
12 import org.jdom.Document;
13 import org.jdom.Element;
14 import org.jdom.JDOMException;
15 import org.jdom.input.SAXBuilder;
16 import org.jdom.xpath.XPath;
17
18 public class ExtractBiTextMethods {
19
20         static PrintWriter reCommentsFile; // Txt-File to recover the comments (1 file per class)
21         static PrintWriter AllCommentsOfProject; // All the comments of a project in 1 file.
22
23         static PrintWriter reCodeLinkComFile; // Txt-File with the linked code of the comments put in the reCommentsFile. (
               Parallel lines)
24         static PrintWriter AllCodeLinkComOfProject; // All the linked code of all the comments of the project (Parallel with
               AllCommentsOfProject).
25
26         /**
27          * Transform File f (XML File) to an org.jdom.Document for applying JDOM/XPath-Function on it.
28          * @param File f
29          * @return org.jdom.Document d
30          * @author <a href="mailto:laurentjakubina@gmail.com">Laurent Jakubina</a>
31          */
32         public static Document prepare(File f){
33
34                 Document d = null;
35                 System.out.println("Prepare: "+f);
36                 try {
37                         /* On crÃ©e une instance de SAXBuilder */
38                         SAXBuilder sxb = new SAXBuilder();
39                         d = sxb.build(f);
40                 } catch (IOException ioe) {
41                         System.out.println("Erreur lors de la lecture du fichier: "+f.getName()+" :"+ioe.getMessage());
42                         ioe.printStackTrace();
43                 } catch (JDOMException ioe){
44                         System.out.println("Erreur lors de la construction du fichier JDOM: " +f.getName()+" :"+ ioe.
                               getMessage());
45                         //ioe.printStackTrace();
46                         return null;
47                 }
48                 return d;
49         }
50
51         /**
52          * Clean the comment by removing some special characters.
53          * @param String comment
54          * @return the comment without some special characters (/**, //, tab, @JavaDoc, etc.)
55          * @author <a href="mailto:laurentjakubina@gmail.com">Laurent Jakubina</a>
```

```
56          */
57          public static String cleanComment(String comment){
58                      comment = comment.replaceAll("/","")       // Remove the char "/".
59                                  .replaceAll("\\*","")       // Remove the char "*".
60                                          .replaceAll("\\n"," ")      // Remove the new lines.
61
62                                              .replaceAll("@return", "RETURN ")
63
64                                                  .replaceAll("(?m)@.*$","")      // Remove the Java-Doc lines.7
65                                                      .replaceAll("(?m)JUnitDoclet.*$","")// Remove the
                                                            JUnitDoclet comments
66                                                          .replaceAll("(?m)^\\s{1,}","") // Remove the
                                                                white space at the beginning of a line.
67                                                              .replaceAll("\\W"," ")
68                                                                  .replaceAll("\\s{2,}"," ")
69                                                  ;
70                      if (comment.matches("^\\W.*$") || comment.matches("^_.*$")){
71                              return "";
72                      } else
73                              return comment;
74          }
75
76          /**
77           * Analyse the parents nodes of the comment (=noeudCourant) to know if the comment
78           * describes a method, a class or some code.
79           * @param Element noeudCourant
80           * @return The type of the comment: CLASS, METHOD OR CODE as a string.
81           * @throws JDOMException
82           * @author <a href="mailto:laurentjakubina@gmail.com">Laurent Jakubina</a>
83           */
84          public static String typeComment(Element noeudCourant)throws JDOMException{
85
86                  XPath xpa = XPath.newInstance("./ancestor::*[position()=2]");
87                  Element temp = (Element) xpa.selectSingleNode(noeudCourant);
88
89                  if (temp.getName().matches("\\w+MethodHeader")){
90                          return "METHOD";
91                  } else {
92                                  if (temp.getName().matches("\\w+ClassDeclaration")){
93                                          return "CLASS";
94                                  } else {
95                                                  return "CODE";
96                                                  }
97                                  }
98          }
99
100         /**
101          * Analyse the parents nodes to extract the header-code corresponding to the comment (=noeudCourant).
102          * @param Element noeudCourant
103          * @return The header-code of the comment (in the noeudCourant).
104          * @author <a href="mailto:laurentjakubina@gmail.com">Laurent Jakubina</a>
105          */
106         public static String HeaderCode(Element noeudCourant)throws JDOMException{
107
108                 XPath xpa = XPath.newInstance("./ancestor::*[position()=2]");
109                 Element temp = (Element) xpa.selectSingleNode(noeudCourant);;
110
111                 if ((temp != null) && temp.getName().matches("\\w+MethodHeader")){
112
```

```
113                     xpa = XPath.newInstance("child::*[1]");
114                     Element temp2 = (Element) xpa.selectSingleNode(temp);
115
116                     Element temp3 = (Element) temp.clone();
117                     temp3.getChild(temp2.getName()).removeChildren("Comment");
118                     return temp3.getValue().replace("\n","").replace(";","");
119
120             } else {
121                     if ((temp != null) && temp.getName().matches("\\w+ClassDeclaration")){
122
123                             xpa = XPath.newInstance("child::*[1]");
124                             Element temp2 = (Element) xpa.selectSingleNode(temp);
125
126                             Element result = (Element) temp.clone();
127                             result.getChild(temp2.getName()).removeChildren("Comment");
128                             result.removeChildren("AClassBody");
129
130                             return result.getValue().replace("\n","").replace(";","");
131
132                             // Remonter jusqu'au noeud ...STATEMENT pour avoir pour le code pour le reste des coms.
133
134                     } else {
135                                     return "";
136
137                     }
138             }
139     }
140
141     /**
142      * In function of a comment of a method (noeudCourant), returns the code of the method, without
143      * the comments and a part of the header ("\\w+modifier"). If the body of the method is empty
144      * (no code - Only comments - Interface), return "". With that, the calling method can filter
145      * this method and don't take it. (no code = no interresting information in this case).
146      * @param Element noeudCourant
147      * @return The code the method.
148      * @throws JDOMException
149      * @author <a href="mailto:laurentjakubina@gmail.com">Laurent Jakubina</a>
150      */
151     public static String BMCode(Element noeudCourant) throws JDOMException{
152
153             XPath xpa = XPath.newInstance("./ancestor::*[position()=3]");
154             Element temp = (Element) xpa.selectSingleNode(noeudCourant);
155             // Ascend to the begin of the method to recover it.
156             if ((temp != null) && temp.getName().matches("\\w+MethodDeclaration")){
157
158                     Element result = (Element) temp.clone();
159                     xpa = XPath.newInstance(".//*"); // Take all the children-nodes = All the method (id, exception,
                            body-code, comments)
160                     List<Element> childsOfMethod = xpa.selectNodes(result);
161                     Iterator<Element> delallcom = childsOfMethod.iterator();
162                     Iterator<Element> i = childsOfMethod.iterator();
163
164                     // Remove all the comments of the body of the method (easier to treat the rest of the body)
165                     while(delallcom.hasNext()){
166                             Element com = delallcom.next();
167                             if ((com != null) && com.getName().matches("Comment")){
168                                     Element com2 = com.getParentElement();
169                                     com2.removeChild(com.getName());
170                             }
```

```
171                     }
172
173                     while(i.hasNext()){
174                             Element temp2 = i.next();
175                             if ((temp2 != null) && temp2.getName().matches("\\w+Modifier")){
176                                     Element temp3 = temp2.getParentElement();
177                                     temp3.removeChild(temp2.getName());
178                     // } else // <AEmptyMethodBody>; </AEmptyMethodBody> -> No Body-Code !
179                     //  if ((temp2 != null) && temp2.getName().matches("AEmptyMethodBody")) {return "";}
180                     //  else // Check if <ABlock> </ABlock> is empty. If yes, no body code too !
181                     //   if ((temp2 != null) && temp2.getName().matches("ABlock")) {
182                     //    if (temp2.getValue().replace("\n","").equals("{ } ")) {return "";} }
183                     }
184                     }
185                     return result.getValue().replace("\n","") // Remove the new lines.
186                                                                     .replaceAll("\\W"," ") // Remove the special
                                                                         characters.
187                                                                     .replaceAll("\\s{2,}"," ") // Remove
                                                                         the white-spaces bigger than 2
                                                                         white-spaces.
188                                     ;
189             } else { return "To Check"; }
190     }
191
192     /**
193      * Take the string s (in this case, it's the body core of a method as string) et split it
194      * according to the White Space (" "). Each word is taken for splitting if necessary.
195      * The splitter used is the Identifiers Splitter : http://splitit.cs.loyola.edu/web-service.html.
196      * @param String s
197      * @return The code of the method with the splitting identifiers.
198      * @author <a href="mailto:laurentjakubina@gmail.com">Laurent Jakubina</a>
199      */
200     public static String SplitIdent(String s){
201
202             String result = new String();
203             result = s;
204     // result = result.replaceAll("[\\W&&[^\\s]]", "").replaceAll("\\s+"," ");
205
206             // Splitter split = new Splitter();
207             // Identifiers Splitter Object: http://splitit.cs.loyola.edu/web-service.html
208
209             if (result.equals("")) {return "";}
210             else {
211                     String[] SplitPhrase = result.split(" ");
212                     List<String> SplitPhraseList = Arrays.asList(SplitPhrase);
213                     Iterator<String> i = SplitPhraseList.iterator();
214
215                     result = new String();
216
217                     while (i.hasNext()){
218                             String StoTreat = (String) i.next();
219
220                             Pattern p = Pattern.compile("[A-Z[^ \\W]]");
221                             // Checks if the word contains capital letters and no special characters.
222                             // If YES, it's maybe a cut to do (Optimisation).
223                             Matcher m = p.matcher(StoTreat);
224
225                             if (m.find()) {
226
```

```
227                             // ----------Online Identifiers Splitter CALLING----------
228                             // try {
229                             //  StoTreat = split.conservativeSplit(StoTreat);
230                             // } catch(IOException e) {System.out.println("exception " + e);}
231                             // ------------------------------------------------------------
232
233                                     String[] PST = StoTreat.split("(?<=\\p{Lower})(?=(\\p{Lu}))");
234                                     List<String> PSTL = Arrays.asList(PST);
235                                     Iterator<String> j = PSTL.iterator();
236                                     StoTreat = new String();
237                                     while (j.hasNext()){
238                                             String recup = j.next();
239                                             if (recup.equals("")){
240                                                     // Do Nothing
241                                             } else {
242                                                     StoTreat += recup+" ";
243                                             }
244                                     }
245                             }
246                             result += StoTreat+" ";
247                     }
248                     return result.replace("  ", " ");
249             }
250     }
251
252     /**
253      * Travel all the (node-)comments from the document to write them in the file.txt/output.
254      * @param org.jdom.Document d
255      * @throws JDOMException
256      * @author <a href="mailto:laurentjakubina@gmail.com">Laurent Jakubina</a>
257      */
258     public static void proceed(Document d) throws JDOMException{
259
260             XPath xpa = XPath.newInstance("/Start/ACompilationUnit//Comment");
261             List<Element> comments = xpa.selectNodes(d);
262             Iterator<Element> i = comments.iterator();
263
264             String comment = ""; // THE comment
265             String type = ""; // The type of the comment : CLASS, METHODE, CODE (Others).
266             //String headercode = ""; // The header code with the comment : (method header, etc).
267             String bmcode = ""; // The code of the body of the method.
268
269             while(i.hasNext()){
270                     Element noeudCourant = (Element) i.next();
271
272                     type = typeComment(noeudCourant);
273
274                     // Updated for treating only the comments/code of the METHODS - Choice of 20/10/2011 {*
275                     if (type.equals("METHOD")){
276
277                             bmcode = BMCode(noeudCourant);
278                             bmcode = SplitIdent(bmcode);
279
280                             //headercode = HeaderCode(noeudCourant);
281
282                             comment = noeudCourant.getValue().trim();
283                             comment = cleanComment(comment);
284                             comment = SplitIdent(comment);
285
```

```
286                                     if (!comment.equals("") && !comment.equals(" ") && !bmcode.equals("")){
287                                             // Remove empty comments because of the "clean" AND Filter the methods (
                                                    take only).
288                                             //System.out.println(type+" : "+comment);
289                                             reCommentsFile.println(/*type+":"+*/comment);
290                                             AllCommentsOfProject.println(/*type+":"+*/comment);
291
292                                             //System.out.println(linkcode);
293                                             reCodeLinkComFile.println(bmcode);
294                                             AllCodeLinkComOfProject.println(bmcode);
295                                     }
296                             }
297                             // *}
298                     }
299                     reCommentsFile.close();
300                     reCodeLinkComFile.close();
301             }
302
303             public static void main(String[] args) {
304
305                     File repository = new File("C:\\Users\\Laurent\\Documents\\Master 2\\Stage Canada\\
                            ToutesLesClassesDeJHotDraw\\ToutesLesClassesParsÃ©es");
306                     int taille = repository.listFiles().length;
307
308                     try {
309                             AllCommentsOfProject = new PrintWriter("C:\\Users\\Laurent\\Documents\\Master 2\\Stage
                                    Canada\\ToutesLesClassesDeJHotDraw\\TousLesCommentairesDeChaqueClasse\\
                                    ALL_COMMENTS.txt");
310                             AllCodeLinkComOfProject = new PrintWriter("C:\\Users\\Laurent\\Documents\\Master 2\\Stage
                                    Canada\\ToutesLesClassesDeJHotDraw\\TousLesCommentairesDeChaqueClasse\\
                                    ALL_LINKED_CODE_OF_COMMENTS.txt");
311                     } catch (FileNotFoundException e2) {
312                             // TODO Auto-generated catch block
313                             e2.printStackTrace();
314                     }
315
316                     for(int i = 0; i < taille; i++){
317
318                             File f = repository.listFiles()[i];
319
320                             //AllCommentsOfProject.println(f.getName());
321
322                             String fname = f.getName().substring(0,f.getName().lastIndexOf("."));
323                             try {
324                                     reCommentsFile = new PrintWriter("C:\\Users\\Laurent\\Documents\\Master 2\\Stage
                                            Canada\\ToutesLesClassesDeJHotDraw\\TousLesCommentairesDeChaqueClasse\\"+
                                            fname+"_COMMENTS.txt");
325                                     reCodeLinkComFile = new PrintWriter("C:\\Users\\Laurent\\Documents\\Master 2\\Stage
                                            Canada\\ToutesLesClassesDeJHotDraw\\TousLesCommentairesDeChaqueClasse\\"+
                                            fname+"_LINKEDCODE.txt");
326                             } catch (FileNotFoundException e1) {
327                                     // TODO Auto-generated catch block
328                                     e1.printStackTrace();
329                             }
330                             try {
331                                     Document d = prepare(f);
332                                     if (d == null){
333                                     } else {
334                                             proceed(d);
```

```
335                             }
336                         } catch (JDOMException e) {
337                             // TODO Auto-generated catch block
338                             e.printStackTrace();
339                         }
340                 }
341                 AllCommentsOfProject.close();
342                 AllCodeLinkComOfProject.close();
343         }
344 }
```

Code Source : GenEchanTrad.java

```
1
2  import java.io.BufferedReader;
3  import java.io.FileInputStream;
4  import java.io.FileNotFoundException;
5  import java.io.IOException;
6  import java.io.InputStream;
7  import java.io.InputStreamReader;
8  import java.io.PrintWriter;
9
10 public class GenEchanTrad {
11
12         static PrintWriter outFileWithTrad;
13
14         public static String mosesTrad(String line){
15
16                 Runtime runtime = Runtime.getRuntime();
17
18                 try {
19
20                         Process moses = runtime.exec(new String[] {"/bin/sh", "-c", "echo '"+line+"' | //usr//bin//moses -f
                               //home//laurent//Bureau//StageUdeMTrainingMoses//5_Corpus18Projects//test//model//moses.
                               ini"});
21                         BufferedReader reader = new BufferedReader(new InputStreamReader(moses.getInputStream()));
22                         String line2 = "";
23
24                         try {
25                                 while((line2 = reader.readLine()) != null) {
26                                         //System.out.println(line2);
27                                         line = line2;
28                                 }
29
30                         } finally { reader.close(); }
31
32                 } catch (IOException e) { e.printStackTrace(); }
33
34                 return line;
35         }
36
37         public static void main(String[] args) {
38
39                 int nbr = 1;
40                 String file ="//home//laurent//Bureau//DiffJHD76-731_LINKEDCODE_V2.txt";
41                 try {
42                         outFileWithTrad = new PrintWriter("//home//laurent//Bureau//GenTrad_V3");
43                 } catch (FileNotFoundException e) {
44                         // TODO Auto-generated catch block
45                         e.printStackTrace();
46                 }
47
48                 try{
49                         InputStream ips = new FileInputStream(file);
50                         InputStreamReader ipsr = new InputStreamReader(ips);
51                         BufferedReader br = new BufferedReader(ipsr);
52                         String line;
53                         while ((line = br.readLine()) != null){
54                                 System.out.println(nbr);
55                                 line = mosesTrad(line);
56                                 outFileWithTrad.println(line);
```

```
57                                  nbr++;
58                          }
59                          br.close();
60                  }
61                  catch (Exception e){
62                          System.out.println(e.toString());
63                  }
64
65                  outFileWithTrad.close();
66          }
67  }
```

MODERNISATION **QUÉBEC**

Des ratés informatiques

■ Le déploiement des systèmes est paralysée et les coûts du projet SAGIR seront dépassés

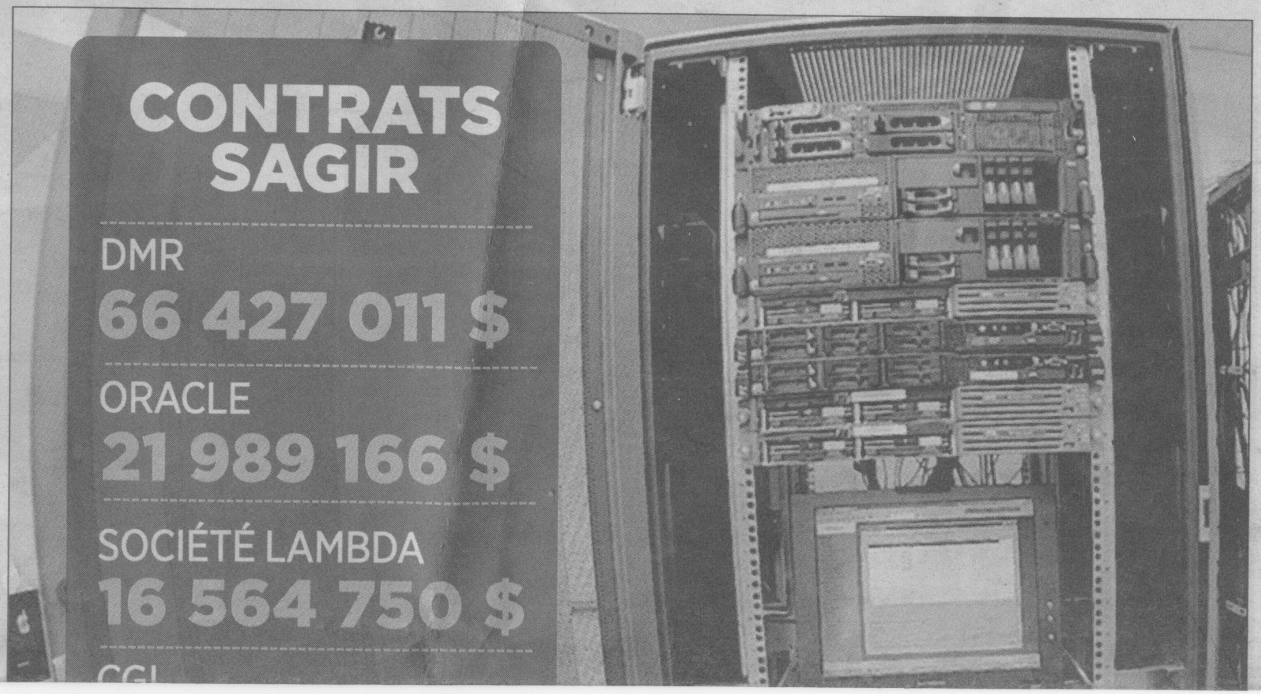
QUÉBEC | Après avoir raté son échéancier à deux reprises, un des plus importants projets informatiques du gouvernement, SAGIR, se retrouve complètement paralysé, a appris le *Journal*.

Rémi Nadeau
Bureau parlementaire
remi.nadeau@quebecormedia.com

Le déploiement de SAGIR, estimé à 450 millions $, est crucial. Il comprend un ensemble de plateformes informatiques devant moderniser les systèmes de gestion des ressources humaines, matérielles et financières de l'administration publique.

Si la livraison de la première phase a respecté son budget de 300 millions $ et l'échéancier de 2008, il en va tout autrement des phases 2 et 3.

Au printemps 2011, l'ex-vérificateur général du Québec, Renaud Lachance, avait signalé que la deuxième livraison, estimée à 103 millions $, n'avait pas été complétée comme prévu en juin 2010.



(CSPQ), qui gère l'important projet, avait alors fixé son nouvel échéancier à juin 2012.

Or, questionné par le *Journal*, le service des communications a admis que, loin d'être déployée, SGR2 fait l'objet d'un réexamen.

« Les besoins d'affaires identifiés en 2006 ont été mis à jour. La livraison 2 fait actuellement l'objet d'une réactualisation de la portée, des coûts et de l'échéancier », nous a-t-on répondu dans un courriel.

En fait, en novembre 2010, la base informatique devant servir d'assise pour le déploiement des fonctionnalités de SGR2 a été installée dans 113 ministères et organismes, mais depuis, le projet devant desservir 75 000 employés s'est enlisé.

Le CSPQ n'est pas en mesure de préciser une nouvelle échéance ni d'écarter des dépassements de coût.

Et la phase 3 ?

Quant à la phase 3, qui doit permettre tant au grand public qu'aux employés actuels de l'administration publique de postuler pour des postes à combler, elle n'a toujours pas été enclenchée.

Un appel d'offres a pourtant été lancé par le CSPQ, mais à la suite de l'évaluation des propositions, il a jugé qu'elles n'étaient pas conformes « aux besoins identifiés ».

Un budget de 20,2 millions a été autorisé pour la troisième étape de SAGIR qui, au total, devait se décliner en sept phases.

Sur son site Web, le CSPQ écrit encore aujourd'hui que les travaux préparatoires de SGR2 et SGR3 sont en cours « en vue d'une implantation en diverses étapes d'ici 2011-2012 ».

Les volets subséquents prévoient la modernisation des systèmes comptables, d'analyse financière et du système de paie.

PHOTO AGENCE QMI, DIDIER DEBUSSCHÈRE

■ Le CPSQ n'est pas en mesure de préciser une nouvelle échéance.

Une fortune en services-conseils

(RN) | Sur l'ensemble des sommes dépensées jusqu'ici dans SAGIR, seulement 11,5 millions $ ont été consacrés à l'acquisition d'équipements informatiques, pendant que cinq firmes se partageaient une impressionnante manne de 197 millions $ en services-conseils.

Questionné à ce sujet, le Centre des services partagés du Québec (CSPQ) a seulement fourni une réponse par courriel, stipulant que « l'infrastructure technologique est unique pour toutes les phases, et que le matériel acquis pour la première phase est toujours utilisé ».

Sur la multiplication des contrats de services professionnels, de conseillers stratégiques, d'expertise et d'accompagnement, le CSPQ plaide qu'elle s'explique par « le nombre d'étapes en cours de développement ainsi que la diversité des expertises requises ».

Le CSPQ, qui gère d'importants projets informatiques gouvernementaux, a lui-même énormément recours aux consultants.

S'il compte sur une équipe de 1 095 personnes pour travailler sur ces projets, il embauche aussi à contrat pas moins de 603 employés à temps complet externe.

PQ désabusé

Informé par le *Journal* des difficultés de déploiement de SAGIR, le député péquiste Sylvain Simard s'est dit convaincu que les travaux de deuxième livraison devront être repris.

« Je connais ce langage. Lorsqu'on dit que le projet fait l'objet d'une réactualisation de portée, c'est qu'on doit recommencer », a soupiré le député de Richelieu.

Il affirme s'être inquiété de l'état d'avancement du projet à deux reprises depuis l'automne dernier, sans jamais avoir reçu de réponse de la présidente du Conseil du trésor, Michelle Courchesne.

« Non, mais les projets informatiques, ils ne pourraient pas en réussir un ? Chaque fois, ça prend l'eau », a commenté M. Simard.